Big Data Programming 2 Lecture Note

Jinheum Kim

CONTENTS

Syllabus

- 교과목명 (학수번호): 빅데이터프로그래밍2 (10934-001)
- 개설 학기: 2017년 2학기
- 강의개요: 본 강좌는 파이선을 써서 빅데이터 기초통계분석을 할 수 있는 역량을 배양하기 위하여 파이선 기본 사용법과 numpy, pandas, matplotlib, statsmodels 등의 라이브러리를 학습하고 이를 써서 프로그래밍 실습을 하고자 함
- 이수구분: 전공필수(빅데이터융한연계정공), 전공선택(응용통계학전공)
- 학점 / 시간: 3 / 3
- 강의실: 글로벌경상관 507호
- 역량구분: 자기주도역량, 글로벌역량, 전문역량
- 평가방법: 상대평가
- 평가비율: 출석 10%, 과제 10%, 중간고사 40%, 기말고가 40%

Syllabus

Week	Topics	Note
1	Python language basics	
2	Built-in data structures, functions, and files 1	
3	Built-in data structures, functions, and files 2	
4	Built-in data structures, functions, and files 3	
5	NumPy basics: arrays and vectorized computations 1	
6	NumPy basics: arrays and vectorized computations 2	
7	NumPy basics: arrays and vectorized computations 3	
8	Mid-term	

Syllabus

Week	Topics	Note
9	Getting stratted with pandas 1	
10	Getting stratted with pandas 2	
11	Data cleaning and preparation	
12	Data wrangling: join, combine, and reshape	
13	Plotting and visualization	
14	Data aggregation and group operations	
15	Final	

Week 1:
Python Language Basics

The Python Interpreter

The Basics

OUTLINE

THE PYTHON INTERPRETER

The Python Interpreter

- Python is an interpreted language
- The standard interactive Python interpreter can be invoked on the command line with the **python** command
  - $ python
  - Python 2.7.2 (default, Oct 4 2011, 20:06:09)
  - [GCC 4.6.1] on linux2
  - Type "help", "copyright", "credits" or "license" for more information.
  - >>>
- The **>>>** you see is the prompt where you'll type expressions
- To exit the Python interpreter and return to the command prompt, you can either type **exit()** or press **Ctrl-D**

The Python Interpreter

- Running Python programs is as simple as calling python with a **.py** file as its first argument
  - Suppose we had created **hello_world.py** with these contents:
    - print 'Hello world'
  - $ python hello_world.py

The Python Interpreter

- By using the **%run** command, **IPython** executes the code in the specified file in the same process, enabling you to explore the results interactively when it's done
  - $ ipython
  - Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul 3 2011, 15:17:51)
  - Type "copyright", "credits" or "license" for more information.
  - IPython 0.12 -- An enhanced Interactive Python.
  - ? -> Introduction and overview of IPython's features.
  - %quickref -> Quick reference.
  - help -> Python's own help system.
  - object? -> Details about 'object', use 'object??' for extra details.

  - In []: %run hello_world.py
  - Hello world

Language Semantics

Scalar Types

Flow Control

THE BASICS

LANGUAGE SEMANTICS

Indentation, not braces

- Python uses whitespace (tabs or spaces) to structure code instead of using braces
- A colon **:** denotes the start of an indented code block after which all of the code must be indented by the same amount until the end of the block
  - for x in array:
    - if x < pivot:
      - less.append(x)
    - else:
      - greater.append(x)

Comments

- Any text preceded by the hash mark (pound sign) # is ignored by the Python interpreter
  - results = []
  - for line in file_handle:
    - # keep the empty lines for now
    - # if len(line) == 0:
    - # continue
    - results.append(line.replace('foo', 'bar'))

Function and object method calls

- Functions are called using parentheses and passing zero or more arguments, optionally assigning the returned value to a variable
  - result = f(x, y, z)
  - g()
- Functions can take both positional and keyword arguments
  - result = f(a, b, c, d=5, e='foo')
- Almost every object in Python has attached functions, known as methods, that have access to the object's internal contents
  - They can be called using the syntax: obj.some_method(x, y, z)

Variables and pass-by-reference

- When assigning a variable in Python, you are creating a reference to the object on the right hand side of the equals sign
  - a = [1, 2, 3]
  - b = a
- In Python, **a** and **b** refer to the same object
  - a.append(4); b
  - Out[]: [1, 2, 3, 4]

Variables and pass-by-reference

- When you pass objects as arguments to a function, you are only passing references; no copying occurs. Python is said to pass by reference
  - def append_element(some_list, element):
    - some_list.append(element)
  - data = [1, 2, 3]
  - append_element(data, 4); data
  - Out[]: [1, 2, 3, 4]

변수명 작명법

- 영문자, 숫자, underscore(_)만 사용 가능
- 영대소문자를 구별함
- 숫자를 첫 글자로 사용할 수 없음
- is, not, if, for 등과 같이 특별한 단어는 사용할 수 없음

Dynamic references, strong types

- Object references in Python have no type associated with them
  - a = 5
  - type(a)
  - Out[]: int
  - a = 'foo'
  - type(a)
  - Out[]: str

Dynamic references, strong types

- '5' + 5
  - TypeError: cannot concatenate 'str' and 'int' objects
- In some languages, such as Visual Basic, the string '5' might get implicitly converted to an integer, thus yielding 10
- Yet in other languages, such as JavaScript, the integer 5 might be casted to a string, yielding the concatenated string '55'

Dynamic references, strong types

- Python is considered a strongly-typed language, which means that every object has a specific type, and implicit conversions will occur only in certain obvious circumstances
  - a = 4.5
  - b = 2
  - a / b
  - Out[]: 2.25

Dynamic references, strong types

- You can check that an object is an instance of a particular type using the **isinstance** function
  - a = 5
  - isinstance(a, int)
  - Out[]: True
- **isinstance** can accept a tuple of types if you want to check that an object's type is among those present in the tuple
  - a = 5; b = 4.5
  - isinstance(a, (int, float))
  - Out[]: True
  - isinstance(b, (int, float))
  - Out[]: True

Attributes and methods

- Objects in Python typically have both attributes and methods. Both of them are accessed via the syntax **obj.attribute_name**
  - a = 'foo'; a.<Tab>

```
a.capitalize  a.format      a.isupper     a.rindex      a.strip
a.center      a.index       a.join        a.rjust       a.swapcase
a.count       a.isalnum     a.ljust       a.rpartition  a.title
a.decode      a.isalpha     a.lower       a.rsplit      a.translate
a.encode      a.isdigit     a.lstrip      a.rstrip      a.upper
a.endswith    a.islower     a.partition   a.split       a.zfill
a.expandtabs  a.isspace     a.replace     a.splitlines
a.find        a.istitle     a.rfind       a.startswith
```

“Duck” typing

- You may not care about the type of an object but rather only whether it has certain methods
- For example, you can verify that an object is iterable if it implemented the iterator protocol. For many objects, this means it has a **__iter__** magic method, though an alternative and better way to check is to try using the **iter** function:
  - def isiterable(obj):
    - try:
      - iter(obj)
      - return True
    - except TypeError: # not iterable
      - return False

“Duck” typing

- This function would return True for strings as well as most Python collection types
  - isiterable('a string')
  - Out[]: True
  - isiterable([1, 2, 3])
  - Out[]: True
  - isiterable(5)
  - Out[]: False

Imports

- In Python a module is simply a .py file containing function and variable definitions along with such things imported from other .py files
  - # some_module.py
  - PI = 3.14159
  - def f(x):
    - return x + 2
  - def g(a, b):
    - return a + b
- If we wanted to access the variables and functions defined in some_module.py, from another file in the same directory we could do
  - import some_module
  - result = some_module.f(5)
  - pi = some_module.PI

Imports

- Or equivalently:
  - from some_module import f, g, PI
  - result = g(5, PI)
- By using the **as** keyword you can give imports different variable names
  - import some_module as sm
  - from some_module import PI as pi, g as gf
  - r1 = sm.f(pi)
  - r2 = gf(6, pi)

Binary operators and comparisons

- Most of the binary math operations and comparisons are as you might expect
  - 5 – 7
  - Out[]: -2
  - 12 + 21.5
  - Out[]: 33.5
  - 5 <= 2
  - Out[]: False

Binary operators and comparisons

Table A-1. Binary operators

Operation	Description
`a + b`	Add a and b
`a - b`	Subtract b from a
`a * b`	Multiply a by b
`a / b`	Divide a by b
`a // b`	Floor-divide a by b, dropping any fractional remainder
`a ** b`	Raise a to the b power
`a & b`	`True` if both a and b are `True`. For integers, take the bitwise AND.
`a | b`	`True` if either a or b is `True`. For integers, take the bitwise OR.
`a ^ b`	For booleans, `True` if a or b is `True`, but not both. For integers, take the bitwise EXCLUSIVE-OR.
`a == b`	`True` if a equals b
`a != b`	`True` if a is not equal to b
`a <= b, a < b`	`True` if a is less than (less than or equal) to b
`a > b, a >= b`	`True` if a is greater than (greater than or equal) to b
`a is b`	`True` if a and b reference same Python object
`a is not b`	`True` if a and b reference different Python objects

Binary operators and comparisons

- a & b: For integers, take the bitwise AND
  - a=2, b=3이라 하면, a는 이진법으로는 10이고 3은 11이므로 2의 자릿수는 1 AND 1=1이고, 1의 자릿수는 0 AND 1=0이므로 결과 는 이진법으로 10. 따라서 십진법으로 2
  - a = 3; b = 2; c = a & b; c
  - Out[]: 2
- a | b: For integers, take the bitwise OR
  - a=2, b=3이라 하면, 2의 자릿수는 1 | 1=1이고, 1의 자릿수는 0 | 1=1이므로 결과는 이집법으로 11. 따라서 십진법으로 3
  - a = 3; b = 2; c = a | b; c
  - Out[]: 3

Binary operators and comparisons

- a ^ b: For integers, take the bitwise EXCLUSIVE-OR
  - a=2, b=3이라 하면, 2의 자릿수는 1 ^ 1=0이고, 1의 자릿수는 0 ^ 1=1이므로 결과는 이진법으로 01. 따라서 십진법으로 31
  - a = 2; b = 3; c = a ^ b; c
  - Out[]: 1

Binary operators and comparisons

- To check if two references refer to the same object, use the **is** keyword
- **is not** is also perfectly valid if you want to check that two objects are not the same
  - a = [1, 2, 3]
  - b = a
  - # Note, the list function always creates a new list
  - c = list(a)
  - a is b
  - Out[]: True
  - a is not c
  - Out[]: True

Binary operators and comparisons

- Note this is not the same thing as comparing with == , because in this case we have
  - a == c
  - Out[]: True
- A very common use of **is** and **is not** is to check if a variable is **None** , since there is only one instance of **None**
  - a = None
  - a is None
  - Out[]: True

Mutable and immutable objects

- Most objects in Python are mutable, such as lists, dicts, NumPy arrays, or most user-defined types
- This means that the object or values that they contain can be modified
  - a_list = ['foo', 2, [4, 5]]
  - a_list[2] = (3, 4); a_list
  - Out[]: ['foo', 2, (3, 4)]
- Others, like strings and tuples, are immutable
  - a_tuple = (3, 5, (4, 5))
  - a_tuple[1] = 'four'
    - TypeError: 'tuple' object does not support item assignment

SCALAR TYPES

Python scalar types

- Python has a small set of built-in types for handling numerical data, strings, boolean (True or False) values, and dates and times
- Date and time handling will be discussed separately as these are provided by the **datetime** module in the standard library

Python scalar types

Table A-2. Standard Python Scalar Types

Type	Description
None	The Python "null" value (only one instance of the None object exists)
`str`	String type. ASCII-valued only in Python 2.x and Unicode in Python 3
`unicode`	Unicode string type
`float`	Double-precision (64-bit) floating point number. Note there is no separate `double` type.
`bool`	A `True` or `False` value
`int`	Signed integer with maximum value determined by the platform.
`long`	Arbitrary precision signed integer. Large `int` values are automatically converted to `long`.

Python scalar types

- Python 3.xx 버전에서는 long type이 int type으로 통일됨
  - ival = 17239871; type(ival)
  - Out[]: int
  - ival2 = ival ** 6; type(ival2)
  - Out[]: int
  - ival2
  - Out[]: 26254519291092456596965462913230729701102721
- 또한 unicode type이 str type으로 통일됨
  - type('hello')
  - Out[]: str
  - type(u'hello')
  - Out[]: str

Numeric types

- The primary Python types for numbers are **int** and **float**
  - ival = 17239871
  - type(ival)
  - Out[]: int
  - ival2 = ival ** 6
  - type(ival2)
  - Out[]: int
  - ival2
  - Out[]: 26254519291092456596965462913230729701102721

Numeric types

- Floating point numbers are represented with the Python **float** type
- They can also be expressed using scientific notation
  - fval = 7.243
  - fval2 = 6.78e-5
- In Python 3, integer division not resulting in a whole number will always yield a floating point number
  - 3 / 2
  - Out[]: 1.5

Numeric types

- In Python 2.7 and below, you can enable this behavior by default by putting the following cryptic-looking statement at the top of your module
  - from __future__ import division
  - 3/2
  - Out[]: 1.5
- Without this in place, you can always explicitly convert the denominator into a floating point number
  - 3 / float(2)
  - Out[]: 1.5
- To get C-style integer division, use the floor division operator //
  - 3 // 2
  - Out[]: 1

Strings

- You can write string literal using either single quotes ' or double quotes "
  - a = 'one way of writing a string'
  - b = "another way “
  - print(a)
    - one way of writing a string
  - print(b)
    - another way
- For multiline strings with line breaks, you can use triple quotes, either ''' or """ :
  - c = """
  - This is a longer string that
  - spans multiple lines
  - """“
  - print(c)
    - This is a longer string that
    - spans multiple lines

Strings

- Python strings are immutable; you cannot modify a string without creating a new string
  - a = 'this is a string'
  - a[10] = 'f'
    - TypeError: 'str' object does not support item assignment
  - b = a.replace('string', 'longer string'); b
  - Out[]: 'this is a longer string'

Strings

- Many Python objects can be converted to a string using the **str** function:
  - a = 5.6
  - s = str(a); s
  - Out[]: '5.6'
- Strings are a sequence of characters and therefore can be treated like other sequences, such as lists and tuples:
  - s = 'python'
  - list(s)
  - Out[]: ['p', 'y', 't', 'h', 'o', 'n']
  - s[:3]
  - Out[]: 'pyt'

Strings

- The backslash character \ is an escape character, meaning that it is used to specify special characters like newline **\n** or unicode characters. To write a string literal with backslashes, you need to escape them
  - s = '12\\34'
  - print(s)
    - 12\34

Strings

- If you have a string with a lot of backslashes and no special characters, you might find this a bit annoying. Fortunately you can preface the leading quote of the string with **r** which means that the characters should be interpreted as is
  - s = r'this\has\no\special\characters'; s
  - Out[]: 'this\\has\\no\\special\\characters'

Strings

- Adding two strings together concatenates them and produces a new string
  - a = 'this is the first half '
  - b = 'and this is the second half'
  - a + b
  - Out[]: 'this is the first half and this is the second half'

Strings

- String templating or formatting is another important topic. Strings with a % followed by one or more format characters is a target for inserting a value into that string
  - template = '%.2f %s are worth $%d'
- %s means to format an argument as a string, %.2f a number with 2 decimal places, and %d an integer. To substitute arguments for these format parameters, use the binary operator % with a tuple of values
  - template % (4.5560, 'Argentine Pesos', 1)
  - Out[]: '4.56 Argentine Pesos are worth $1'

Booleans

- The two boolean values in Python are written as **True** and **False.** Comparisons and other conditional expressions evaluate to either **True** or **False.** Boolean values are combined with the **and** and **or** keywords
  - True and True
  - Out[]: True
  - False or True
  - Out[]: True

Booleans

- Almost all built-in Python tops and any class defining the **__nonzero__** magic method have a **True** or **False** interpretation in an **if** statement

- a = [1, 2, 3]
- if a:
  - print ('I found something!')
- I found something!
- b = []
- if not b:
  - print ('Empty! ')
- Empty!

Booleans

- Most objects in Python have a notion of true- or false-ness. For example, empty sequences (lists, dicts, tuples, etc.) are treated as **False** if used in control flow. You can see exactly what boolean value an object coerces to by invoking **bool** on it
  - bool([]), bool([1, 2, 3])
  - Out[]: (False, True)
  - bool('Hello world!'), bool('')
  - Out[]: (True, False)
  - bool(0), bool(1)
  - Out[]: (False, True)

Type casting

- The **str**, **bool**, **int** and **float** types are functions which can be used to cast values to those types

- s = '3.14159'
- fval = float(s)
- type(fval)
- Out[]: float
- int(fval)
- Out[]: 3
- bool(fval)
- Out[]: True
- bool(0)
- Out[]: False

None

- **None** is the Python null value type. If a function does not explicitly return a value, it implicitly returns **None**
  - a = None
  - a is None
  - Out[]: True
  - b = 5
  - b is not None
  - Out[]: True
- **None** is also a common default value for optional function arguments
  - def add_and_maybe_multiply(a, b, c=None):
    - result = a + b
    - if c is not None:
      - result = result * c
    - return result

Dates and times

- The built-in Python **datetime** module provides **datetime**, **date**, and **time** types. The **datetime** type combines the information stored in date and time and is the most commonly used
  - from datetime import datetime, date, time
  - dt = datetime(2011, 10, 29, 20, 30, 21)
  - dt.day
  - Out[]: 29
  - dt.minute
  - Out[]: 30

Dates and times

- Given a datetime instance, you can extract the equivalent date and time objects by calling methods on the datetime of the same name
  - dt.date()
  - Out[]: datetime.date(2011, 10, 29)
  - dt.time()
  - Out[]: datetime.time(20, 30, 21)
- The **strftime** method formats a datetime as a string
  - dt.strftime('%m/%d/%Y %H:%M')
  - Out[]: '10/29/2011 20:30'
- Strings can be converted into datetime objects using the **strptime** function
  - datetime.strptime('20091031', '%Y%m%d')
  - Out[]: datetime.datetime(2009, 10, 31, 0, 0)

Dates and times

- When aggregating of otherwise grouping time series data, it will be useful to replace fields of a series of datetimes, for example replacing the minute and second fields with zero, producing a new object
  - dt.replace(minute=0, second=0)
  - Out[]: datetime.datetime(2011, 10, 29, 20, 0)

Dates and times

- The difference of two datetime objects produces a **datetime.timedelta** type
  - dt2 = datetime(2011, 11, 15, 22, 30)
  - delta = dt2 - dt; delta
  - Out[]: datetime.timedelta(17, 7179)
  - type(delta)
  - Out[]: datetime.timedelta
- Adding a timedelta to a datetime produces a new shifted datetime
  - dt
  - Out[]: datetime.datetime(2011, 10, 29, 20, 30, 21)
  - dt + delta
  - Out[]: datetime.datetime(2011, 11, 15, 22, 30)

FLOW CONTROL

if, elif, and else

- The **if** statement checks a condition which, if True, evaluates the code in the block that follows
  - if x < 0:
    - print ("It's negative")
- An **if** statement can be optionally followed by one or more **elif** blocks and a catch-all **else** block if all of the conditions are **False**
  - if x < 0:
    - print ("It's negative")
  - elif x == 0:
    - print('Equal to zero')
  - elif 0 < x < 5:
    - print('Positive but smaller than 5')
  - else:
    - print('Positive and larger than or equal to 5')

for loops

- **for** loops are for iterating over a collection (like a list or tuple) or an iterator. The standard syntax for a for loop is:
  - for value in collection:
- A for loop can be advanced to the next iteration, skipping the remainder of the block, using the **continue** keyword
  - sequence = [1, 2, None, 4, None, 5]
  - total = 0
  - for value in sequence:
    - if value is None:
      - continue
    - total += value

for loops

- A for loop can be exited altogether using the **break** keyword
  - sequence = [1, 2, 0, 4, 6, 5, 2, 1]
  - total_until_5 = 0
  - for value in sequence:
    - if value == 5:
      - break
    - total_until_5 += value

while loops

- A **while** loop specifies a condition and a block of code that is to be executed until the condition evaluates to False or the loop is explicitly ended with **break**
  - x = 256
  - total = 0
  - while x > 0:
    - if total > 500:
      - break
    - total += x
    - x = x // 2

pass

- **pass** is the “no-op” statement in Python. It can be used in blocks where no action is to be taken
  - if x < 0:
    - print('negative!')
  - elif x == 0:
    - # TODO: put something smart here
    - pass
  - else:
    - print('positive!')
- It’s common to use **pass** as a place-holder in code while working on a new piece of functionality
  - def f(x, y, z):
    - # TODO: implement this function!
    - pass

Exception handling

- Handling Python errors or exceptions gracefully is an important part of building robust programs
- In data analysis applications, many functions only work on certain kinds of input. As an example, Python's **float** function is capable of casting a string to a floating point number, but fails with ValueError on improper inputs
  - float('1.2345')
  - Out[]: 1.2345
  - float('something')
    - ValueError: could not convert string to float: something

Exception handling

- Suppose we wanted a version of float that fails gracefully, returning the input argument. We can do this by writing a function that encloses the call to float in a try/except block

- def attempt_float(x):
  - try:
    - return float(x)
  - except:
    - return x
- attempt_float('1.2345')
- Out[]: 1.2345
- attempt_float('something')
- Out[]: 'something'

Exception handling

- You might notice that **float** can raise exceptions other than ValueError
  - float((1, 2))
    - TypeError: float() argument must be a string or a number
- You might want to only suppress ValueError, since a TypeError might indicate a legitimate bug in your program
  - def attempt_float(x):
    - try:
      - return float(x)
    - except ValueError:
      - return x
  - attempt_float((1, 2))
    - TypeError: float() argument must be a string or a number

Exception handling

- You can catch multiple exception types by writing a tuple of exception types
- def attempt_float(x):
  - try:
    - return float(x)
  - except (TypeError, ValueError):
    - return x

Exception handling

- You may not want to suppress an exception, but you want some code to be executed regardless of whether the code in the **try** block succeeds or not. To do this, use **finally**
  - f = open(path, 'w')
  - try:
    - write_to_file(f)
  - finally:
    - f.close()
- You can have code that executes only if the **try** block succeeds using **else** block
  - f = open(path, 'w')
  - try:
    - write_to_file(f)
  - except:
    - print('Failed')
  - else:
    - print ('Succeeded')
  - finally:
    - f.close()

range

- The **range** function produces a list of evenly-space integers
  - range(10)
  - Out[]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
- Both a start, end, and step can be given
  - range(0, 20, 2)
  - Out[]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
- This snippet sums all numbers from 0 to 9999 that are multiples of 3 or 5
- sum = 0
- for i in range(10000):
  - if i % 3 == 0 or i % 5 == 0:
    - sum += i

Ternary Expressions

- A ternary expression in Python allows you combine an if-else block which produces a value into a single line or expression
- The syntax for this in Python is
  - value = true-expr if condition else false-expr
- x = 5
- 'Non-negative' if x >= 0 else 'Negative'
- Out[]: 'Non-negative'

Week 2-4:
Built-in data structures, functions, and files

Tuple
List
Built-in Sequence Functions
Dict
Set
List, Set, and Dict Comprehensions

DATA STRUCTURES AND SEQUENCES

TUPLE

Tuple

- A tuple is a one-dimensional, fixed-length, immutable sequence of Python objects
- The easiest way to create one is with a comma-separated sequence of values
  - tup = 4, 5, 6; tup
  - Out[]: (4, 5, 6)
- When defining tuples in more complicated expressions, it's necessary to enclose the values in parentheses. For example,
  - nested_tup = (4, 5, 6), (7, 8); nested_tup
  - Out[]: ((4, 5, 6), (7, 8))

Tuple

- Any sequence or iterator can be converted to a tuple by invoking **tuple**
  - tuple([4, 0, 2])
  - Out[]: (4, 0, 2)
  - tup = tuple('string'); tup
  - Out[]: ('s', 't', 'r', 'i', 'n', 'g')
- Elements can be accessed with square brackets []. Note that sequences are 0-indexed in Python
  - tup[0]
  - Out[]: 's'

Tuple

- While the objects stored in a tuple may be mutable themselves, once created it's not possible to modify which object is stored in each slot:
  - tup = tuple(['foo', [1, 2], True])
  - tup[2] = False
    - TypeError: 'tuple' object does not support item assignment
  - tup[1].append(3); tup
  - Out[]: ('foo', [1, 2, 3], True)
- Tuples can be concatenated using the + operator to produce longer tuples
  - (4, None, 'foo') + (6, 0) + ('bar',)
  - Out[]: (4, None, 'foo', 6, 0, 'bar')
- Multiplying a tuple by an integer has the effect of concatenating together many copies of the tuple
  - ('foo', 'bar') * 4
  - Out[]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')

Tuple

- (4, None, 'foo') + (6, 0) + ('bar')
  - TypeError: can only concatenate tuple (not "str") to tuple(4, None, 'foo') + (6, 0) + ('bar')
  - 에라메세지가 발생한 이유는 세 번째 object를 tuple로 인식하지 못하고 str로 잘못 인식 했기 때문에. 해결방법은 콤마(,)를 삽입하면 됨. 따라서 tuple의 원소가 1개 뿐 일 때는 이와 같은 방법으로 하면 tuple로 인식됨.
- (4, None, 'foo') + (6, 0) + ('bar')
- Out[]: (4, None, 'foo', 6, 0, 'bar')

Unpacking tuples

- If you try to assign to a tuple-like expression of variables, Python will attempt to unpack the value on the right-hand side of the equals sign
  - tup = (4, 5, 6)
  - a, b, c = tup; b
  - Out[]: 5
- Sequences with nested tuples can be unpacked
  - tup = 4, 5, (6, 7)
  - a, b, (c, d) = tup; d
  - Out[]: 7

Unpacking tuples

- It's easy to swap variable names
  - tmp = a; a = b; b = tmp

    ⟺
  - b, a = a, b
- One of the most common uses of variable unpacking when iterating over sequences of tuples or lists
  - seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
  - for a, b, c in seq:
    - print(a); print(b); print(c)

```
1
2
3
4
5
6
7
8
9
```

Tuple methods

- Since the size and contents of a tuple cannot be modified, it is very light on instance methods. One particularly useful one is **count**, which counts the number of occurrences of a value
  - a = (1, 2, 2, 2, 3, 4, 2)
  - a.count(2)
  - Out[]: 4

LIST

List

- Lists are variable-length and their contents can be modified. They can be defined using square brackets [] or using the **list** type function
  - a_list = [2, 3, 7, None]
  - tup = ('foo', 'bar', 'baz')
  - b_list = list(tup); b_list
  - Out[]: ['foo', 'bar', 'baz']
  - b_list[1] = 'peekaboo'; b_list
  - Out[]: ['foo', 'peekaboo', 'baz']

Adding and removing elements

- Elements can be appended to the end of the list with the **append** method
  - b_list.append('dwarf'); b_list
  - Out[]: ['foo', 'peekaboo', 'baz', 'dwarf']
- Using **insert** you can insert an element at a specific location in the list
  - b_list.insert(1, 'red'); b_list
  - Out[]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']

Adding and removing elements

- The inverse operation to **insert** is **pop**, which removes and returns an element at a particular index
  - b_list.pop(2)
  - Out[]: 'peekaboo'; b_list
  - Out[]: ['foo', 'red', 'baz', 'dwarf']
- Elements can be removed by value using **remove**, which locates the first such value and removes it
  - b_list.append('foo'); b_list
  - Out[]: ['foo', 'peekaboo', 'baz', 'foo']
  - b_list.remove('foo'); b_list
  - Out[]: ['peekaboo', 'baz', 'foo']

Concatenating and combining lists

- Adding two lists together with + concatenates them
  - [4, None, 'foo'] + [7, 8, (2, 3)]
  - Out[]: [4, None, 'foo', 7, 8, (2, 3)]
- If you have a list already defined, you can append multiple elements to it using the **extend** method
  - x = [4, None, 'foo']
  - x.extend([7, 8, (2, 3)])
  - x
  - Out[]: [4, None, 'foo', 7, 8, (2, 3)]

Concatenating and combining lists

- Using **extend** to append elements to an existing list is usually preferable
  - everything = []
  - for chunk in list_of_lists:
    - everything.extend(chunk)
- is faster than
  - everything = []
  - for chunk in list_of_lists:
    - everything = everything + chunk

Sorting

- A list can be sorted in-place by calling its **sort** function
  - a = [7, 2, 5, 1, 3]
  - a.sort(); a
  - Out[]: [1, 2, 3, 5, 7]
- **sort** has a few options that will come in handy. One is the ability to pass a secondary sort key
  - b = ['saw', 'small', 'He', 'foxes', 'six']
  - b.sort(key=len); b
  - Out[]: ['He', 'saw', 'six', 'small', 'foxes']

Binary search and maintaining a sorted list

- The built-in **bisect** module implements binary-search and insertion into a sorted list. **bisect.bisect** finds the location where an element should be inserted to keep it sorted, while **bisect.insort** actually inserts the element into that location

- import bisect
- c = [1, 2, 2, 2, 3, 4, 7]
- bisect.bisect(c, 2)
- Out[]: 4
- bisect.bisect(c, 5)
- Out[]: 6
- bisect.insort(c, 6); c
- Out[]: [1, 2, 2, 2, 3, 4, 6, 7]

Binary search and maintaining a sorted list

- list에 python 객체를 추가할 때, 대소비교가 가능한 경우 bisect 모듈의 insort 함수를 이용하면 정렬된 상태를 유지하면서 객체를 추가할 수 있음
- 따라서 "**bisect.insort** actually inserts the element into that location"의 뜻은 "추가될 개체의 자리를 찾고 그 자리에 명시한 값을 추가한다."는 뜻임

Slicing

- You can select sections of list-like types (arrays, tuples, NumPy arrays) by using slice notation, which consists of **start:stop** passed to the indexing operator []
  - seq = [7, 2, 3, 7, 5, 6, 0, 1]
  - seq[1:5]
  - Out[]: [2, 3, 7, 5]
- Slices can also be assigned to with a sequence
  - seq[3:4] = [6, 3]; seq
  - Out[]: [7, 2, 3, 6, 3, 5, 6, 0, 1]

Slicing

- While element at the **start** index is included, the **stop** index is not included, so that the number of elements in the result is **stop – start**
- Either the **start** or **stop** can be omitted in which case they default to the start of the sequence and the end of the sequence, respectively
  - seq[:5]
  - Out[]: [7, 2, 3, 6, 3]
  - seq[3:]
  - Out[415]: [6, 3, 5, 6, 0, 1]
- Negative indices slice the sequence relative to the end:
  - seq[-4:]
  - Out[]: [5, 6, 0, 1]
  - seq[-6:-2]
  - Out[]: [6, 3, 5, 6]

Slicing

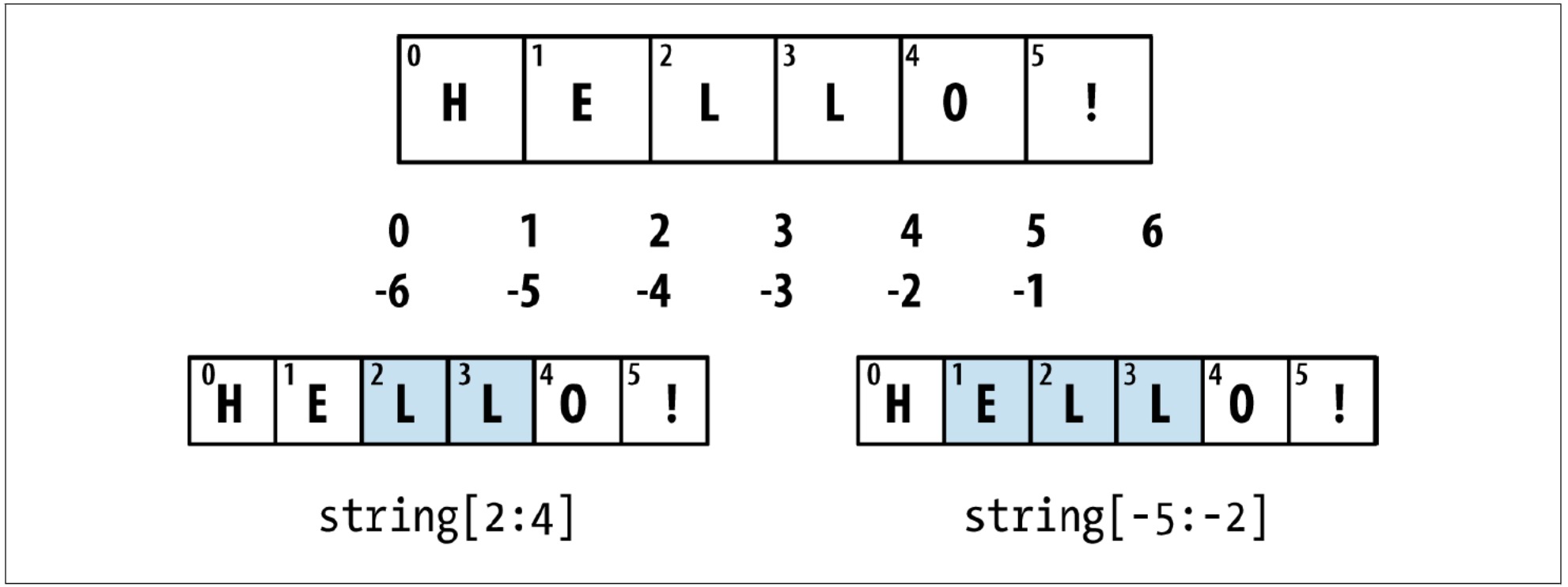


Figure A-2. Illustration of Python slicing conventions

Slicing

- A **step** can be used after a second colon to take every other element
  - seq[::2]
  - Out[]: [7, 3, 3, 6, 1]
- A clever use of this is to pass -1 which has the useful effect of reversing a list or tuple
  - seq[::-1]
  - Out[]: [1, 0, 6, 5, 3, 6, 3, 2, 7]

BUILT-IN SEQUENCE FUNCTIONS

enumerate()

- It's common to keep track of the index of the current item when iterating over a sequence
  - i = 0
  - for value in collection:
    - # do something with value
    - i += 1
- Python has a built-in function **enumerate** which returns a sequence of (i, value) tuples
  - for i, value in enumerate(collection):
    - # do something with value

enumerate()

- When indexing data, a useful pattern that uses **enumerate** is computing a **dict** mapping the values of a sequence to their locations in the sequence
  - some_list = ['foo', 'bar', 'baz']
  - mapping = dict((v, i) for i, v in enumerate(some_list)); mapping
  - Out[]: {'bar': 1, 'baz': 2, 'foo': 0}

sorted()

- The **sorted** function returns a new sorted list from the elements of any sequence
  - sorted([7, 1, 2, 6, 0, 3, 2])
  - Out[]: [0, 1, 2, 2, 3, 6, 7]
  - sorted('horse race')
  - Out[]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
- A common pattern for getting a sorted list of the unique elements in a sequence is to combine sorted with set
  - sorted(set('this is just some string'))
  - Out[]: [' ', 'e', 'g', 'h', 'i', 'j', 'm', 'n', 'o', 'r', 's', 't', 'u']

zip()

- **zip** "pairs" up the elements of a number of lists, tuples, or other sequences, to create a list of tuples
  - seq1 = ['foo', 'bar', 'baz']
  - seq2 = ['one', 'two', 'three']
  - list(zip(seq1, seq2))
  - Out[]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
- **zip** can take an arbitrary number of sequences, and the number of elements it produces is determined by the shortest sequence
  - seq3 = [False, True]
  - zip(seq1, seq2, seq3)
  - Out[]: [('foo', 'one', False), ('bar', 'two', True)]

zip()

- A common use of **zip** is for simultaneously iterating over multiple sequences, possibly combined with **enumerate**
  - for i, (a, b) in enumerate(zip(seq1, seq2)):
  - print('%d: %s, %s' % (i, a, b))
  - 0: foo, one
  - 1: bar, two
  - 2: baz, three

zip()

- Given a zipped sequence, **zip** can be applied to unzip the sequence
  - pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'), ('Schilling', 'Curt')]
  - first_names, last_names = zip(*pitchers)
  - first_names
  - Out[]: ('Nolan', 'Roger', 'Schilling')
  - last_names
  - Out[]: ('Ryan', 'Clemens', 'Curt')
- Note that zip(*seq) ⟺ zip(seq[0], seq[1],…,seq[len(seq)-1])

reversed()

- **reversed** iterates over the elements of a sequence in reverse order
  - list(reversed(range(10)))
  - Out[]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

DICT

Dict

- **dict** is the most important built-in Python data structure. It is a flexibly-sized collection of key-value pairs, where key and value are Python objects. One way to create one is by using curly braces {} and using colons to separate keys and values
  - empty_dict = {}
  - d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}; d1
  - Out[]: {'a': 'some value', 'b': [1, 2, 3, 4]}

Dict

- Elements can be accessed and inserted or set using the same syntax as accessing elements of a list or tuple
  - d1[7] = 'an integer'; d1
  - Out[]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
  - d1['b']
  - Out[]: [1, 2, 3, 4]

Dict

- You can check if a **dict** contains a key using the same syntax as with checking whether a list or tuple contains a value
  - 'b' in d1
  - Out[]: True
- Values can be deleted either using the **del** keyword or the **pop** method
  - d1[5] = 'some value'
  - d1['dummy'] = 'another value'
  - del d1[5]
  - ret = d1.pop('dummy'); ret
  - Out[]: 'another value'

Dict

- The **keys** and **values** method give you lists of the keys and values, respectively
  - d1.keys()
  - Out[]: ['a', 'b', 7]
  - d1.values()
  - Out[]: ['some value', [1, 2, 3, 4], 'an integer']
- One dict can be merged into another using the **update** method
  - d1.update({'b' : 'foo', 'c' : 12}); d1
  - Out[]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}

Creating dicts from sequences

- It's common to end up with two sequences that you want to pair up element-wise in a dict
  - mapping = {}
  - for key, value in zip(key_list, value_list):
    - mapping[key] = value
- Since a dict is a collection of 2-tuples, it should be no shock that the **dict** type function accepts a list of 2-tuples
  - mapping = dict(zip(range(5), reversed(range(5)))); mapping
  - Out[]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}

Default values

- It's very common to have logic like:
  - if key in some_dict:
    - value = some_dict[key]
  - else:
    - value = default_value
- The dict methods **get** and **pop** can take a default value to be returned, so that the above **if-else** block can be written simply as
  - value = some_dict.get(key, default_value)
- **get** by default will return **None** if the key is not present, while **pop** will raise a exception

Default values

- For example, you could imagine categorizing a list of words by their first letters as a dict of lists
  - words = ['apple', 'bat', 'bar', 'atom', 'book']
  - by_letter = {}
  - for word in words:
    - letter = word[0]
    - if letter not in by_letter: # 키가 dict에 있는지를 판단
      - by_letter[letter] = [word] # 새로운 키와 값을 dict에 추가함
    - else:
      - by_letter[letter].append(word) # 기존 키에 값만 추가함
  - by_letter
  - Out[]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}

Default values

- The **setdefault** dict method is for precisely this purpose. The **if-else** block above can be rewritten as
  - by_letter.setdefault(letter, []).append(word)
- The built-in **collections** module has a useful class, **defaultdict**, which makes this even easier
  - from collections import defaultdict
  - by_letter = defaultdict(list)
  - for word in words:
    - by_letter[word[0]].append(word)

Valid dict key types

- While the values of a dict can be any Python object, the keys have to be immutable objects like scalar types (int, float, string) or tuples. You can check whether an object is hashable with the **hash** function
  - hash('string')
  - Out[79]: 2014442006951182849
  - hash((1, 2, (2, 3)))
  - Out[]: 10976365022763477782
  - hash((1, 2, [2, 3])) # fails because lists are mutable
    - TypeError: unhashable type: 'list'
- To use a list as a key, an easy fix is to convert it to a tuple
  - d = {}
  - d[tuple([1, 2, 3])] = 5; d
  - Out[]: {(1, 2, 3): 5}

SET

Set

- A set is an unordered collection of unique elements. You can think of them like dicts, but keys only, no values
- A set can be created in two ways: via the **set** function or using a set literal with curly braces {}
  - set([2, 2, 2, 1, 3, 3])
  - Out[]: set([1, 2, 3])
  - {2, 2, 2, 1, 3, 3}
  - Out[]: set([1, 2, 3])
- Sets support mathematical set operations like union, intersection, difference, and symmetric difference

Set

- a = {1, 2, 3, 4, 5}
- b = {3, 4, 5, 6, 7, 8}
- a | b # union (or)
- Out[]: {1, 2, 3, 4, 5, 6, 7, 8}
- a & b # intersection (and)
- Out[]: {3, 4, 5}
- a - b # difference
- Out[]: set([1, 2])
- a ^ b # symmetric difference (xor)
- Out[]: {1, 2, 6, 7, 8}

- You can also check if a set is a subset of or a superset of another set
  - a_set = {1, 2, 3, 4, 5}
  - {1, 2, 3}.issubset(a_set)
  - Out[]: True
  - a_set.issuperset({1, 2, 3})
  - Out[]: True
- As you might guess, sets are equal if their contents are equal
  - {1, 2, 3} == {3, 2, 1}
  - Out[]: True

Set

Table A-3. Python Set Operations

Function	Alternate Syntax	Description
`a.add(x)`	N/A	Add element x to the set a
`a.remove(x)`	N/A	Remove element x from the set a
`a.union(b)`	`a | b`	All of the unique elements in a and b.
`a.intersection(b)`	`a & b`	All of the elements in *both* a and b.
`a.difference(b)`	`a - b`	The elements in a that are not in b.
`a.symmetric_difference(b)`	`a ^ b`	All of the elements in a or b but *not both*.
`a.issubset(b)`	N/A	`True` if the elements of a are all contained in b.
`a.issuperset(b)`	N/A	`True` if the elements of b are all contained in a.
`a.isdisjoint(b)`	N/A	`True` if a and b have no elements in common.

LIST, SET, AND DICT COMPREHENSIONS

List, Set, and Dict Comprehensions

- List comprehensions are one of the most-loved Python language features. They allow you to form a new list by filtering the elements of a collection and transforming the elements passing the filter in one expression
- [expr for val in collection if condition]

- This is equivalent to the following **for loop:**
  - result = []
  - for val in collection:
    - if condition:
      - result.append(expr)
- For example,
  - strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
  - [x.upper() for x in strings if len(x) > 2]
  - Out[]: ['BAT', 'CAR', 'DOVE', 'PYTHON']

List, Set, and Dict Comprehensions

- Set and dict comprehensions are a natural extension, producing sets and dicts in a idiomatically similar way instead of lists
  - dict_comp = {key-expr : value-expr for value in collection if condition}
- A set comprehension looks like the equivalent list comprehension except with curly braces instead of square brackets
  - set_comp = {expr for value in collection if condition}

List, Set, and Dict Comprehensions

- Like list comprehensions, set and dict comprehensions are just syntactic sugar, but they similarly can make code both easier to write and read
  - unique_lengths = {len(x) for x in strings}; unique_lengths
  - Out[88]: {1, 2, 3, 4, 6}

List, Set, and Dict Comprehensions

- As a simple dict comprehension example, we could create a lookup map of these strings to their locations in the list
  - loc_mapping = {val : index for index, val in enumerate(strings)}; loc_mapping
  - Out[]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
- Note that this dict could be equivalently constructed by
  - loc_mapping = dict((val, idx) for idx, val in enumerate(strings))

Nested list comprehensions

- Suppose we have a list of lists containing some boy and girl names:
  - all_data = [['Tom', 'Billy', 'Jefferson', 'Andrew', 'Wesley', 'Steven', 'Joe'], ['Susie', 'Casey', 'Jill', 'Ana', 'Eva', 'Jennifer', 'Stephanie']]
- Suppose we wanted to get a single list containing all names with two or more e's in them
  - names_of_interest = []
  - for names in all_data:
    - enough_es = [name for name in names if name.count('e') >= 2]
    - names_of_interest.extend(enough_es)
  - names_of_interest
  - Out[]: ['Jefferson', 'Wesley', 'Steven', 'Jennifer', 'Stephanie']

Nested list comprehensions

- You can wrap this whole operation up in a single nested list comprehension, which will look like:
  - result = [name for names in all_data for name in names if name.count('e') >= 2]; result
  - Out[]: ['Jefferson', 'Wesley', 'Steven', 'Jennifer', 'Stephanie']

Nested list comprehensions

- The **for** parts of the list comprehension are arranged according to the order of nesting, and any filter condition is put at the end as before
  - some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
  - flattened = [x for tup in some_tuples for x in tup]; flattened
  - Out[]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
- Keep in mind that the order of the for expressions would be the same if you wrote a nested for loop instead of a list comprehension
  - flattened = []
  - for tup in some_tuples:
    - for x in tup:
      - flattened.append(x)

Nested list comprehensions

- You can have many levels of nesting, though if you have more than two or three levels of nesting you should probably start to question your data structure design
  - [[x for x in tup] for tup in some_tuples]
  - Out[]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

Namespaces, Scope, and Local Functions
Returning Multiple Values
Functions Are Objects
Anonymous (lambda) Functions
Closures: Functions that Return Functions
Extended Call Syntax with *args, **kwargs
Currying: Partial Argument Application
Generators

FUNCTIONS

NAMESPACES, SCOPE, AND LOCAL FUNCTIONS

Functions

- Functions are declared using the **def** keyword and returned from using the **return** keyword
  - def my_function(x, y, z=1.5):
    - if z > 1:
      - return z * (x + y)
    - else:
      - return z / (x + y)
- There is no issue with having multiple return statements
- If the end of a function is reached without encountering a **return** statement, **None** is returned

Functions

- Each function can have some number of positional arguments and some number of keyword arguments
- Keyword arguments are most commonly used to specify default values or optional arguments
  - x and y are positional arguments while z is a keyword argument
  - my_function(5, 6, z=0.7)
  - Out[]: 0.0636363636363636
  - my_function(3.14, 7, 3.5)
  - Out[]: 35.49
  - my_function(3.14, 7)
  - Out[]: 15.21
- The keyword arguments must follow the positional arguments (if any)
- You can specify keyword arguments in any order

Namespaces, Scope, and Local Functions

- Functions can access variables in two different scopes: global and local
- An alternate and more descriptive name describing a variable scope in Python is a namespace
- Any variables that are assigned within a function by default are assigned to the local namespace
- The local namespace is created when the function is called and immediately populated by the function's arguments. After the function is finished, the local namespace is destroyed

Namespaces, Scope, and Local Functions

- def func():
  - a = []
  - for i in range(5):
    - a.append(i)
  - print(a)
- func()
  - [0, 1, 2, 3, 4]
- a = []
- def func():
  - for i in range(5):
    - a.append(i)
- func(); a
- Out[]: [0, 1, 2, 3, 4]

- Assigning global variables within a function is possible, but those variables must be declared as global using the **global** keyword
  - a = None
  - def bind_a_variable():
    - global a
    - a = []
  - bind_a_variable(); a
  - Out[]: []

Namespaces, Scope, and Local Functions

- Functions can be declared anywhere, and there is no problem with having local functions that are dynamically created when a function is called
- def outer_function(x, y, z):
  - def inner_function(a, b, c):
    - pass
  - pass

Namespaces, Scope, and Local Functions

```
def outer(a,b):
    def inner(c,d):
        return c+d
    return inner(a,b), inner(3,4)
outer(1,2)
Out[]: (3, 7)
```

- 내부 함수(inner())는 코드에서 외부함수(outer())의 return 문장을 만날 때 수행됨. 즉 외부함수가 전달받은 a와 b의 값을 내부함수 c와 d에 각각 전달함. 또한, c에 3, d에 4를 전달하면 내부함수를 수행하고 그 계산 결과를 반환함

Returning Multiple Values

- def f():
  - a = 5
  - b = 6
  - c = 7
  - return a, b, c
- a, b, c = f()
- return_value = f() #tuple

- def f():
  - a = 5
  - b = 6
  - c = 7
  - return {'a' : a, 'b' : b, 'c' : c}
- return_value = f() #dict

Functions Are Objects

- states = [' Alabama ', 'Georgia!', 'Georgia', 'georgia', 'FlOrIda', 'south carolina##', 'West virginia?']
- import re # Regular expression module
- def remove_punctuation(value):
  - return re.sub('[!#?]', '', value)
- clean_ops = [str.strip, remove_punctuation, str.title]
- def clean_strings(strings, ops):
  - result = []
  - for value in strings:
    - for function in ops:
      - value = function(value)
    - result.append(value)
  - return result
- clean_strings(states, clean_ops)
- Out[]: ['Alabama', 'Georgia', 'Georgia', 'Georgia', 'Florida', 'South Carolina', 'West Virginia']

Anonymous (lambda) Functions

- Python has support for anonymous or lambda functions, which are really just simple functions consisting of a single statement, the result of which is the return value
- They are defined using the **lambda** keyword

- def short_function(x):
  - return x * 2

↔

- equiv_anon = lambda x: x * 2

- def apply_to_list(some_list, f):
  - return [f(x) for x in some_list]
- ints = [4, 0, 1, 5, 6]
- apply_to_list(ints, lambda x: x * 2)

↔

- [x * 2 for x in ints]

Anonymous (lambda) Functions

- Python has support for anonymous or lambda functions, which are really just simple functions consisting of a single statement, the result of which is the return value
- They are defined using the **lambda** keyword

- def short_function(x):
  - return x * 2

↔

- equiv_anon = lambda x: x * 2

- def apply_to_list(some_list, f):
  - return [f(x) for x in some_list]
- ints = [4, 0, 1, 5, 6]
- apply_to_list(ints, lambda x: x * 2)

↔

- [x * 2 for x in ints]

Closures: Functions that Return Functions

- A closure is any dynamically-generated function returned by another function
- The key property is that the returned function has access to the variables in the local namespace where it was created
- The difference between a closure and a regular Python function is that the closure continues to have access to the namespace (the function) where it was created, even though that function is done executing

Closures: Functions that Return Functions

```
def outer(arg):
    def inner():
        return "arg is '%s'" % arg
    return inner
c=outer('a')
c()
Out[]: "arg is 'a' "
```

- inner() 함수는 outer() 함수가 전달받은 arg 변수를 알고 있음
- 코드에서 'return inner' 문장은 (호출되지 않은) inner 함수의 복사본을 반환
- 외부 함수(outer)에 의해 동적으로 생성되고, 그 함수의 변수 값(arg)을 알고 있는 함수(inner) 를 closure라고 함

Closures: Functions that Return Functions

- def make_watcher():
  - have_seen = {}
  - def has_been_seen(x):
    - if x in have_seen:
      - return True
    - else:
      - have_seen[x] = True
      - return False
  - return has_been_seen

- watcher = make_watcher()
- vals = [5, 6, 1, 5, 1, 6, 3, 5]
- [watcher(x) for x in vals]
- Out[]: [False, False, False, True, True, True, False, True]

Closures: Functions that Return Functions

- def make_counter():
  - count = [0]
  - def counter():
    - # increment and return the current count
    - count[0] += 1
    - return count[0]
  - return counter
- counter2 = make_counter()
- counter2()
- Out[]: 1

- def format_and_pad(template, space):
  - def formatter(x):
    - return (template % x).rjust(space)
  - return formatter
- fmt = format_and_pad('%.4f', 15)
- fmt(1.756)
- Out[]: '   1.7560'

Extended Call Syntax with *args, **kwargs

- ***args**: parameter 몇개를 받을지 모르는 경우에 사용. tuple 형태로 전달됨
- def print_param(*args):
  - print(args)
  - for p in args:
    - print(p)
- print_param('a', 'b')
  - ('a', 'b')
  - a
  - b

- ****kwargs**: parameter name을 같이 보낼 수 있음. dict 형태로 전달됨
- def print_param2(**kwargs):
  - print(kwargs)
  - print(kwargs.keys())
  - print(kwargs.values())
  - for name, value in kwargs.items():
    - print("%s : %s" % (name, value))
- print_param2(first = 'a', second = 'b')
  - {'first': 'a', 'second': 'b'}
  - dict_keys(['first', 'second'])
  - dict_values(['a', 'b'])
  - first : a
  - second : b

Extended Call Syntax with *args, **kwargs

- def say_hello_then_call_f(f, *args, **kwargs):
  - print('args is', args)
  - print('kwargs is', kwargs)
  - print("Hello! Now I'm going to call %s" % f)
  - return f(*args, **kwargs)
- def g(x, y, z=1):
  - return (x + y) / z

- say_hello_then_call_f(g, 1, 2, z=5.)
  - args is (1, 2)
  - kwargs is {'z': 5.0}
  - Hello! Now I'm going to call <function g at 0x2dd5cf8>
- Out[]: 0.6

Currying: Partial Argument Application

- Currying means deriving new functions from existing ones by partial argument application
  - from functools import partial
  - def add_numbers(x, y):
    - return x + y
  - add_five = lambda y: add_numbers(5, y)
  - add_five(5)
  - Out[]: 10
- The built-in **functools** module can simplify this process using the **partial** function
  - add_five = partial(add_numbers, 5)
  - add_five(10)
  - Out[]: 15

Generators

- A generator is a simple way to construct a new iterable object. Whereas normal functions execute and return a single value, generators return a sequence of values lazily, pausing after each one until the next one is requested
- To create a generator, use the **yield** keyword instead of **return** in a function

- def squares(n=10):
  - print ('Generating squares from 1 to %d' % (n ** 2))
  - for i in range(1, n + 1):
    - yield i ** 2
- for x in squares() :
  - print (x)
- Generating squares from 0 to 100 1 4 9 16 25 36 49 64 81 100

Generators

- def squares(n=10):
  - for i in range(1, n + 1):
    - yield i ** 2
- g = squares(2)
- print(next(g))
  - 1
- print(next(g))
  - 4
- print(next(g))
  - StopIteration

- def gen():
- yield 'one'
- yield 'two'
- g = gen()
- print(next(g)) # one
- print(next(g)) # two
- 대용량 자료 처리 등은 메모리에 모두 올려놓고 할 수 없음. 그런 경우 한 줄 씩 읽은 뒤 generator를 이용한 반복처리를 하면 편리함

Generator expressions

- This is a generator analogue to list, dict and set comprehensions; to create one, enclose with parenthesis instead of brackets
- gen = (x ** 2 for x in range(100))
- sum(gen)
- Out[]: 328350

- Generator expressions can be used inside any Python function
- sum(x ** 2 for x in range(100))
- Out[]: 328350
- dict((i, i **2) for i in range(5))
- Out[]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

itertools module

- The standard library **itertools** module has a collection of generators for many common data algorithms
- For example, **groupby** takes any sequence and a function; this groups consecutive elements in the sequence by return value of the function
  - groupby(iterable[, keyfunc]): generates (key, sub-iterator) for each unique key

- import itertools
- first_letter = lambda x: x[0]
- names = ['Alan', 'Adam', 'Steven']
- for letter, names in itertools.groupby(names, first_letter):
  - print (letter, list(names))# names is a generator
- A ['Alan', 'Adam']
- S ['Steven']

itertools module

- Make an iterator that returns consecutive keys and groups from the iterable (group by에 의해 iterator가 다시 생성됨). The key is a function computing a key value for each element. Generally, the iterable needs to already be sorted on the same key function (키 값에 따라 정렬 되 있어야).

itertools module

- from itertools import groupby as gby
- list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), 'wombat', 'mongoose', 'malloo', 'camel']
- c = gby(list_things, key=lambda x: x[0]) # 키 값에 따라 정렬이 안 된 상태
- dic = {}
- for k, v in c:
- dic[k] = list(v)
- dic
- Out[]: # 원하는 결과를 얻지 못함
  - {'c': ['camel'],
  - 'd': ['dog', 'donkey'],
  - 'g': ['goat'],
  - 'm': ['mongoose', 'malloo'],
  - 'persons': [('persons', 'man', 'woman')],
  - 'w': ['wombat']}

itertools module

- sorted_list = sorted(list_things, key = lambda x: x[0]) # string의 첫 글자를 기준으로 정렬
- c = gby(sorted_list, key=lambda x: x[0]) #키 값에 따라 정렬된 상태임
- dic = {}
- for k, v in c:
- dic[k] = list(v)

- dic
- Out[]: #원하는 결과를 얻음
  - {'c': ['cow', 'cat', 'camel'],
  - 'd': ['dog', 'donkey'],
  - 'g': ['goat'],
  - 'm': ['mulato', 'mongoose', 'malloo'],
  - 'persons': [('persons', 'man', 'woman')],
  - 'w': ['wombat']}

FILES AND THE OPERATING SYSTEM

Files and the operating system

- To open a file for reading or writing, use the built-in **open** function with either a relative or absolute file path
  - path = 'tmp.txt'
  - f = open(path)
- By default, the file is opened in read-only mode 'r'. We can treat the file handle f like a list and iterate over the lines
  - for line in f:
    - pass
- If we had typed f = open(path, 'w'), a new file at tmp.txt would have been created, overwriting any one in its place

Files and the operating system

Table A-5. Python file modes

Mode	Description
`r`	Read-only mode
`w`	Write-only mode. Creates a new file (deleting any file with the same name)
`a`	Append to existing file (create it if it does not exist)
`r+`	Read and write
`b`	Add to mode for binary files, that is `'rb'` or `'wb'`
`U`	Use universal newline mode. Pass by itself `'U'` or appended to one of the read modes like `'rU'`

Files and the operating system

- To write text to a file, you can use either the file's **write** or **writelines** methods
- For example, we could create a version of prof_mod.py with no blank lines
- with open('tmp2.txt', 'w') as handle:
  - handle.writelines(x for x in open(path) if len(x) > 1)
- open('tmp.txt').readlines()
- Out[]:
- ['Sue\\xc3\\xb1a el rico en su riqueza\n',
- 'que m\\xc3\\xa1s cuidados le ofrece;']

Files and the operating system

Table A-6. Important Python file methods or attributes

Method	Description
`read([size])`	Return data from file as a string, with optional `size` argument indicating the number of bytes to read
`readlines([size])`	Return list of lines in the file, with optional `size` argument
`readlines([size])`	Return list of lines (as strings) in the file
`write(str)`	Write passed string to file.
`writelines(strings)`	Write passed sequence of strings to the file.
`close()`	Close the handle
`flush()`	Flush the internal I/O buffer to disk
`seek(pos)`	Move to indicated file position (integer).
`tell()`	Return current file position as integer.
`closed`	`True` is the file is closed.

Week 5-7:
NumPy Basics: Arrays and Vectorized Computations

OUTLINE

The NumPy ndarray: A Multidimensional Array Object
Universal Functions: Fast Element-wise Array Functions
Data Processing Using Arrays
File Input and Output with Arrays
Linear Algebra
Random Number Generation

THE NUMPY NDARRAY: A MULTIDIMENSIONAL ARRAY OBJECT

The NumPy ndarray: A Multidimensional Array Object

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

- impot numpy as np
- data = np.random.randn(2, 3); data

```
array([[ 0.10099615,  1.10600325, -0.01543286],
       [ 3.57694914, -1.54670335,  0.40441843]])
```

- data.shape
- Out[]: (2, 3)
- data.dtype
- Out[]: dtype('float64')

Creating ndarrays

- The easiest way to create an array is to use the **array** function
- data1 = [6, 7.5, 8, 0, 1]
- arr1 = np.array(data1); arr1
- Out[]: array([6. , 7.5, 8. , 0. , 1.])

- data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
- arr2 = np.array(data2); arr2

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

- arr2.ndim
- Out[]: 2
- arr2.dtype
- Out[]: dtype('int64')

Creating ndarrays

- To create a higher dimensional array with these methods, pass a tuple for the shape
- np.zeros(10)
- Out[]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
- np.zeros((3, 6))

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

- It's not safe to assume that np.empty will return an array of all zeros. In many cases, it will return uninitialized garbage values
- np.empty((2, 3, 2))

```
array([[[  4.94065646e-324,   9.88131292e-324],
        [  1.48219694e-323,   1.97626258e-323],
        [  2.47032823e-323,   2.96439388e-323]],

       [[  3.45845952e-323,   3.95252517e-323],
        [  4.44659081e-323,   4.94065646e-323],
        [  5.43472210e-323,   5.92878775e-323]]])
```

Creating ndarrays

- **arange** is an array-valued version of the built-in Python range function
- np.arange(15)
- Out[]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])

Array creation functions

Table 4-1. Array creation functions

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default.
`asarray`	Convert input to ndarray, but do not copy if the input is already an ndarray
`arange`	Like the built-in `range` but returns an ndarray instead of a list.
`ones, ones_like`	Produce an array of all 1's with the given shape and dtype. `ones_like` takes another array and produces a ones array of the same shape and dtype.
`zeros, zeros_like`	Like `ones` and `ones_like` but producing arrays of 0's instead
`empty, empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`eye, identity`	Create a square N x N identity matrix (1's on the diagonal and 0's elsewhere)

Data Types for ndarrays

- arr1 = np.array([1, 2, 3], dtype=np.float64); arr1.dtype
- Out[]: dtype('float64')
- arr2 = np.array([1, 2, 3], dtype=np.int32); arr2.dtype
- Out[]: dtype('int32')

- You can explicitly convert or cast an array from one dtype to another using ndarray's **astype** method
- arr = np.array([1, 2, 3, 4, 5]); arr.dtype
- Out[]: dtype('int64')
- float_arr = arr.astype(np.float64); float_arr.dtype
- Out[]: dtype('float64')

Data Types for ndarrays

- If I cast some floating point numbers to be of integer dtype, the decimal part will be truncated
- arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1]); arr
- Out[]: array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
- arr.astype(np.int32)
- Out[]: array([3, -1, -2, 0, 12, 10], dtype=int32)

- Should you have an array of strings representing numbers, you can use **astype** to convert them to numeric form
- numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
- numeric_strings.astype(float)
- Out[]: array([1.25, -9.6 , 42.])

Operations between Arrays and Scalars

- Any arithmetic operations between equal-size arrays applies the operation elementwise
- arr = np.array([[1., 2., 3.], [4., 5., 6.]]); arr
- arr * arr

```
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
array([[  1.,   4.,   9.],
       [ 16.,  25.,  36.]])
```

- Arithmetic operations with scalars are propagating the value to each element
- 1 / arr

```
array([[ 1.        ,  0.5       ,  0.33333333],
       [ 0.25      ,  0.2       ,  0.16666667]])
```

Basic Indexing and Slicing

- If you assign a scalar value to a slice, the value is propagated to the entire selection
- arr = np.arange(10); arr
- Out[]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
- arr[5:8]
- Out[]: array([5, 6, 7])
- arr[5:8] = 12; arr
- Out[]: array([0, 1, 2, 3, 4, 12, 12, 12, 8, 9])

- Array slices are views on the original array. The data is not copied, and any modifications to the view will be reflected in the source array
- arr_slice = arr[5:8]; arr_slice[1] = 12345; arr
- Out[]: array([0, 1, 2, 3, 4, 12, 12345, 12, 8, 9])
- arr_slice[:] = 64; arr
- Out[]: array([0, 1, 2, 3, 4, 64, 64, 64, 8, 9])

Basic Indexing and Slicing

- In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays
- arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
- arr2d[2]
- Out[]: array([7, 8, 9])

- Individual elements can be accessed recursively. Or you can pass a comma-separated list of indices
- arr2d[0][2]
- Out[]: 3
- arr2d[0, 2]
- Out[]: 3

Basic Indexing and Slicing

- In multidimensional arrays, if you omit later indices, the returned object will be a lowerdimensional ndarray consisting of all the data along the higher dimensions
- arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
- arr3d[0]

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

- old_values = arr3d[0].copy()
- arr3d[0] = 42; arr3d
- arr3d[0] = old_values; arr3d

```
array([[[42, 42, 42],
        [42, 42, 42]],

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

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

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

Indexing with slices

- arr[1:6] # one-dimensional objects
- Out[]: array([1, 2, 3, 4, 64])
- arr2d[:2]
- Out[]: array([[1, 2, 3], [4, 5, 6]])
- arr2d[:2, 1:]
- Out[]: array([[2, 3], [5, 6]])

- By mixing integer indexes and slices, you get lower dimensional slices
- arr2d[1, :2]
- Out[]: array([4, 5])
- arr2d[2, :1]
- Out[]: array([7])
- arr2d[:, :1]
- Out[]: array([[1], [4], [7]])
- arr2d[:2, 1:] = 0
- Out[]: array([[1, 0, 0], [4, 0, 0], [7, 8, 9]])

Indexing with slices

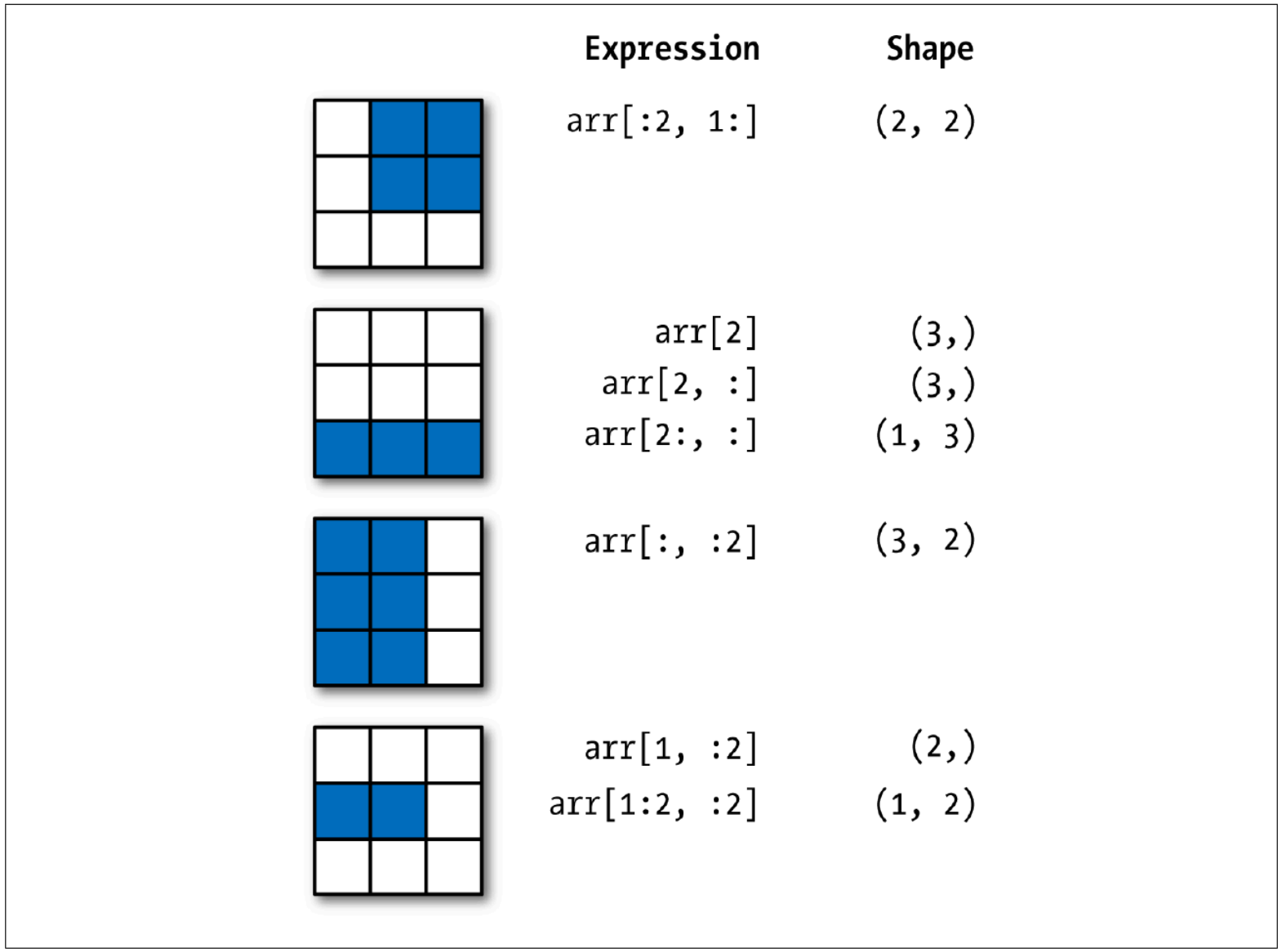


Figure 4-2. Two-dimensional array slicing

Boolean Indexing

- names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
- names == 'Bob' # comparing names with the string 'Bob' yields a boolean array
- Out[]: array([True, False, False, True, False, False, False], dtype=bool)

- data = np.random.randn(7, 4); data
- data[names == 'Bob'] # This boolean array can be passed when indexing the array

```
array([[-0.71746169,  0.02538885,  0.0585284 , -0.26380737],
       [ 0.14710699, -0.26689117, -0.29669834, -0.08477229],
       [-1.84276284, -2.2243701 , -0.90596755,  0.33676928],
       ...,
       [-0.35976402,  1.13092972,  1.04861946,  1.27384805],
       [ 0.07730046, -0.5258066 , -0.29787134, -0.35627773],
       [-0.04941188, -1.80427638, -1.01117   ,  0.31988545]])
```

```
array([[-0.71746169,  0.02538885,  0.0585284 , -0.26380737],
       [-0.70843362,  1.05128121,  0.36287406,  0.42304197]])
```

Boolean Indexing

- data[names == 'Bob', 2:]
- mask = (names == 'Bob') | (names == 'Will'); mask
- Out[]: array([True, False, True, True, True, False, False], dtype=bool)

```
array([[ 0.0585284 , -0.26380737],
       [ 0.36287406,  0.42304197]]
```

- data[mask]
- data[names != 'Joe'] = 7; data

```
array([[-0.71746169,  0.02538885,  0.0585284 , -0.26380737],
       [-1.84276284, -2.2243701 , -0.90596755,  0.33676928],
       [-0.70843362,  1.05128121,  0.36287406,  0.42304197],
       [-0.35976402,  1.13092972,  1.04861946,  1.27384805]])


array([[ 7.        ,  7.        ,  7.        ,  7.        ],
       [ 0.14710699, -0.26689117, -0.29669834, -0.08477229],
       [ 7.        ,  7.        ,  7.        ,  7.        ],
       ...,
       [ 7.        ,  7.        ,  7.        ,  7.        ],
       [ 0.07730046, -0.5258066 , -0.29787134, -0.35627773],
       [-0.04941188, -1.80427638, -1.01117   ,  0.31988545]])
```

Fancy Indexing

- Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays
- arr = np.empty((8, 4))
- for i in range(8):
  - arr[i] = i
- arr

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

Fancy Indexing

- arr = np.arange(32).reshape((8, 4)); arr

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       ...,
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

- arr[[4, 3, 0, 6]]
- arr[[1, 5, 7, 2], [0, 3, 1, 2]]
- Out[]: array([4, 23, 29, 10])
- arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]

```
array([[ 4.,  4.,  4.,  4.],
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.],
       [ 6.,  6.,  6.,  6.]])
```

```
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Transposing Arrays and Swapping Axes

- Arrays have the **transpose** method and also the special **T** attribute
- arr = np.arange(15).reshape((3, 5)); arr

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

- arr.T

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

- When doing matrix computations, you will do this using np.dot
- arr = np.random.randn(6, 3)
- np.dot(arr.T, arr)

```
array([[ 5.13272774, -1.4162491 ,  2.99047577],
       [-1.4162491 ,  0.98390511, -0.79302912],
       [ 2.99047577, -0.79302912,  2.97696196]])
```

Transposing Arrays and Swapping Axes

- For higher dimensional arrays, **transpose** will accept a tuple of axis numbers to permute the axes
- arr = np.arange(16).reshape((2, 2, 4)); arr
- arr.transpose((1, 0, 2))

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

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
```

```
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],

       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

- Simple transposing with **.T** is just a special case of swapping axes. ndarray has the method **swapaxes** which takes a pair of axis numbers
- arr.swapaxes(1, 2)

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

       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

UNIVERSAL FUNCTIONS: FAST ELEMENT-WISE ARRAY FUNCTIONS

Universal Functions

- A universal function, or ufunc, is a function that performs elementwise operations on data in ndarrays
- arr = np.arange(10)
- np.sqrt(arr)#unary ufuncs

```
array([ 0.        ,  1.        ,  1.41421356,  1.73205081,  2.        ,
        2.23606798,  2.44948974,  2.64575131,  2.82842712,  3.        ])
```

- x = np.random.randn(8); x
- y = np.random.randn(8); y
- np.maximum(x, y) # binary ufuncs

```
array([-0.44513058, -0.8056935 , -0.08264555, -0.55052684,  0.03751726,
        0.29815088, -1.08195879,  0.16188054])

array([-1.16505261,  0.28640124,  1.48050392,  1.70501522, -1.02380131,
        0.29333142,  0.62645385,  1.83911478])

array([-0.44513058,  0.28640124,  1.48050392,  1.70501522,  0.03751726,
        0.29815088,  0.62645385,  1.83911478])
```

Universal Functions

Table 4-3. Unary ufuncs

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

Universal Functions

Table 4-4. Binary universal functions

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

DATA PROCESSING USING ARRAYS

Data Processing Using Arrays

- Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops
- Suppose we wished to evaluate the function $\sqrt{x^2 + y^2}$ across a regular grid of values
- points = np.arange(-5, 5, 0.01) # 1000 equally spaced points
- xs, ys = np.meshgrid(points, points) # The np.meshgrid function takes two 1D arrays and produces two 2D matrices corresponding to all pairs of (x, y) in the two arrays

Data Processing Using Arrays

- import matplotlib.pyplot as plt
- z = np.sqrt(xs ** 2 + ys ** 2)
- plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
- plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")

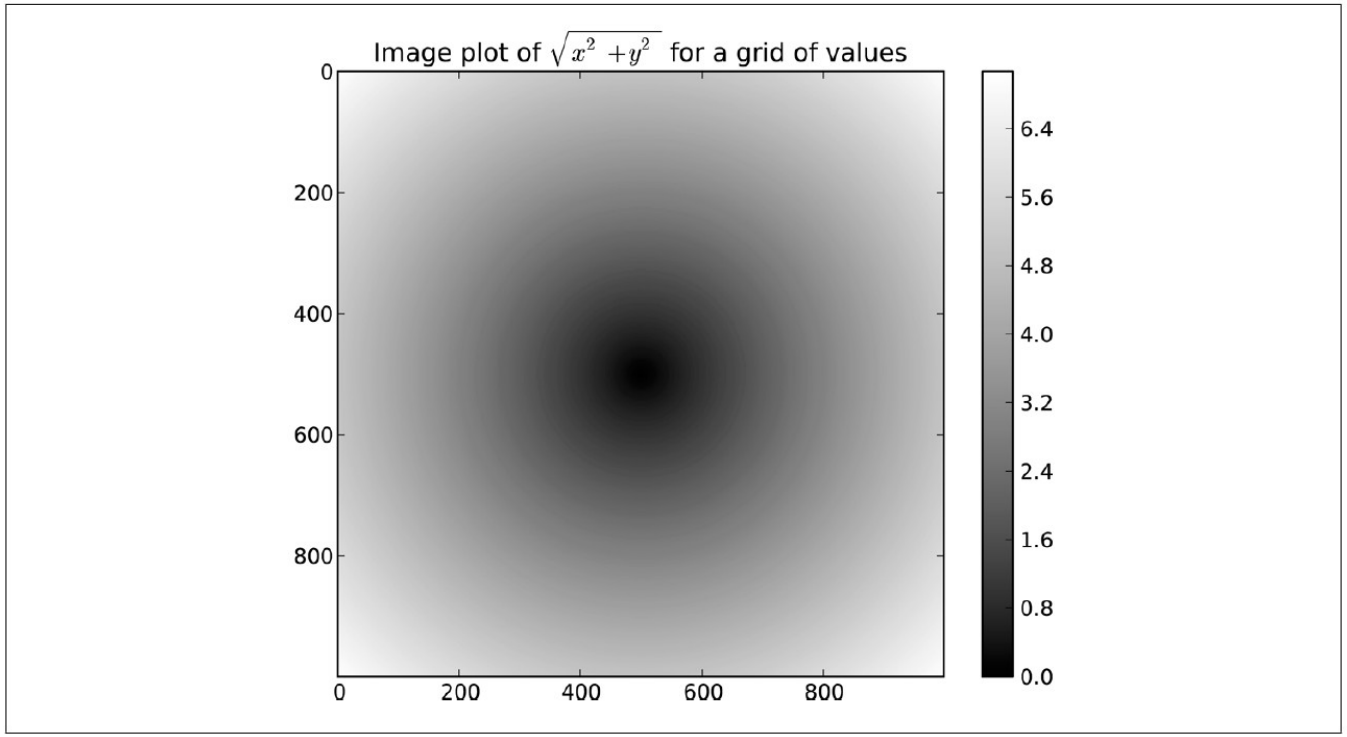


Figure 4-3. Plot of function evaluated on grid

Expressing Conditional Logic as Array Operations

- The **np.where** function is a vectorized version of the ternary expression **x if condition else y** #ternary: 세 개 한 벌의
- xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
- yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
- cond = np.array([True, False, True, True, False])
- result = [(x if c else y) for x, y, c in zip(xarr, yarr, cond)] #큰 배열 처리는 오래 걸림, 다차원 배열 처리 불가능!
- result = np.where(cond, xarr, yarr); result
- Out[]: array([1.1, 2.2, 1.3, 1.4, 2.5])

Expressing Conditional Logic as Array Operations

- Suppose you had a matrix of randomly generated data and you wanted to replace all positive values with 2 and all negative values with -2
- arr = np.random.randn (4, 4); arr
- np.where(arr > 0, 2, -2)

```
array([[-2, -2,  2,  2],
       [ 2,  2,  2, -2],
       [-2,  2,  2, -2],
       [-2, -2,  2,  2]])
```

```
array([[-0.54282832, -1.79057508,  1.37158344,  1.22646394],
       [ 0.6008297 ,  0.63869704,  1.90588798, -0.3655778 ],
       [-0.19488255,  0.44004321,  0.57070293, -0.24935631],
       [-1.49399626, -0.2930051 ,  0.63036972,  0.87538427]])
```

Expressing Conditional Logic as Array Operations

- Use **where** to express more complicated logic
- cond1 = np.array([True, True, False, False])
- cond2 = np.array([True, False, True, False])
- n = 4
- result = []
- for i in range(n):
  - if cond1[i] and cond2[i]:
    - result.append(0)
  - elif cond1[i]:
    - result.append(1)
  - elif cond2[i]:
    - result.append(2)
  - else:
    - result.append(3)
- result
- Out[]: [0, 1, 2, 3]
- result = np.where(cond1 & cond2, 0, np.where(cond1, 1, np.where(cond2, 2, 3)))
- result
- Out[]: array([0, 1, 2, 3])

Mathematical and Statistical Methods

- A set of mathematical functions which compute statistics about an entire array or about the data along an axis are accessible as array methods
- arr = np.random.randn(5, 4) # normally-distributed data
- np.mean(arr)
- Out[]: -0.08326743067411228 3
- np.mean(arr,axis=0)
- Out[]: array([-0.42652313, 0.32055842, 0.28461757, -0.51172259])
- np.mean(arr,axis=1)
- Out[]: array([0.51143058, -0.12222426, -0.53661585, 0.07779331, -0.34672093])

Mathematical and Statistical Methods

- Other methods like **cumsum** and **cumprod** do not aggregate, instead producing an array of the intermediate results
- arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
- np.cumsum(arr, axis=0)

```
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]], dtype=int32)
```

Table 4-5. Basic array statistical methods

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

Methods for Boolean Arrays

- Boolean values are coerced to 1 (True) and 0 (False). *be coerced: 강제로 … 이 되다
- arr = np.random.randn(100)
- np.sum(arr > 0) # Number of positive values
- bools = np.array([False, False, True, False])
- np.any(bools) # Returns True if any of the elements evaluate to True
- Out[]: True
- np.all(bools) # Returns True if all elements evaluate to True.
- Out[]: False

Sorting

- NumPy arrays can be sorted in-place using the **sort** method
- arr = np.random.randn(5); arr
- Out[]: array([0.30905031, 0.19847319, -1.79077161, 1.04988813, 0.44176699])
- np.sort(arr)
- Out[]: array([-1.79077161, 0.19847319, 0.30905031, 0.44176699, 1.04988813])

Sorting

- A quick-and-dirty way to compute the quantiles of an array is to sort it and select the value at a particular rank
- arr = np.random.randn(10); arr

```
array([ 0.99464383, -1.0336272 , -0.67928545,  1.71099   , -1.66619736,
       -2.26439048,  0.92861297,  0.03708037,  0.81780065,  0.9592799 ])

array([-2.26439048, -1.66619736, -1.0336272 , -0.67928545,  0.03708037,
        0.81780065,  0.92861297,  0.9592799 ,  0.99464383,  1.71099   ])
```

- np.sort(arr)
- np.sort(arr)[int(0.1 * len(arr))] # 10% quantile
- Out[]: -1.6661973606839693

Unique and Other Set Logic

- Probably the most commonly used one is **np.unique**, which returns the sorted unique values in an array
- ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
- np.unique(ints)
- Out[]: array([1, 2, 3, 4])

- Another function, **np.in1d**, tests membership of the values in one array in another, returning a boolean array
- values = np.array([6, 0, 0, 3, 2, 5, 6])
- np.in1d(values, [2, 3, 6])
- Out[]: array([True, False, False, True, True, False, True], dtype=bool)

Table 4-6. Array set operations

Method	Description
`unique(x)`	Compute the sorted, unique elements in x
`intersect1d(x, y)`	Compute the sorted, common elements in x and y
`union1d(x, y)`	Compute the sorted union of elements
`in1d(x, y)`	Compute a boolean array indicating whether each element of x is contained in y
`setdiff1d(x, y)`	Set difference, elements in x that are not in y
`setxor1d(x, y)`	Set symmetric differences; elements that are in either of the arrays, but not both

FILE INPUT AND OUTPUT WITH ARRAYS

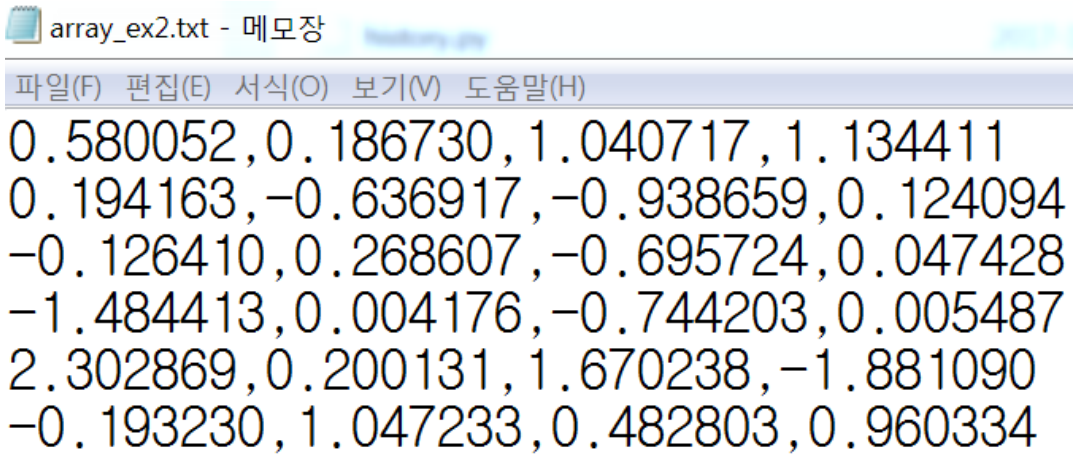
Storing Arrays on Disk in Binary Format

- NumPy is able to save and load data to and from disk either in text or binary format
- **np.save** and **np.load** are the two workhorse functions for efficiently saving and loading array data on disk
- Arrays are saved by default in an uncompressed raw binary format with file extension **.npy**

- arr = np.arange(10)
- np.save('some_array', arr)
- pwd
- Out[]: 'C:\\Users\\jkim'
- np.load('some_array.npy')
- Out[]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Saving and Loading Text Files

- It will be useful to load data into NumPy arrays using **np.loadtxt**
- **np.savetxt** performs the inverse operation: writing an array to a delimited text file
- arr = np.loadtxt('C:\\Users\\jkim\\array_ex2.txt', delimiter=','); arr




```
array([[ 0.580052,  0.18673 ,  1.040717,  1.134411],
       [ 0.194163, -0.636917, -0.938659,  0.124094],
       [-0.12641 ,  0.268607, -0.695724,  0.047428],
       [-1.484413,  0.004176, -0.744203,  0.005487],
       [ 2.302869,  0.200131,  1.670238, -1.88109 ],
       [-0.19323 ,  1.047233,  0.482803,  0.960334]])
```

LINEAR ALGEBRA

Linear Algebra

- from numpy.linalg import inv, qr
- X = np.randm.randn(5, 5)
- mat = X.T.dot(X) # T 메서드는 transpose 함수와 동일
- inv(mat) # $(X'X)^{-1}$ 계산

```
array([[ 3.82611584, -0.4681343 ,  1.13828511, -0.29582443,  2.0404651 ],
       [-0.4681343 ,  0.68592219,  0.51767528,  1.15827818, -0.5258528 ],
       [ 1.13828511,  0.51767528,  1.51511236,  1.17421194,  0.26275403],
       [-0.29582443,  1.15827818,  1.17421194,  2.58446342, -0.66683651],
       [ 2.0404651 , -0.5258528 ,  0.26275403, -0.66683651,  1.32076085]])
```

Table 4-7. Commonly-used numpy.linalg functions

Function	Description
`diag`	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
`dot`	Matrix multiplication
`trace`	Compute the sum of the diagonal elements
`det`	Compute the matrix determinant
`eig`	Compute the eigenvalues and eigenvectors of a square matrix
`inv`	Compute the inverse of a square matrix
`pinv`	Compute the Moore-Penrose pseudo-inverse inverse of a square matrix
`qr`	Compute the QR decomposition
`svd`	Compute the singular value decomposition (SVD)
`solve`	Solve the linear system Ax = b for x, where A is a square matrix
`lstsq`	Compute the least-squares solution to y = Xb

RANDOM NUMBER GENERATION

Random Number Generation

- The **numpy.random** module supplements the built-in Python random with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions
- Python's built-in **random** module, by contrast, only samples one value at a time. **numpy.random** is well over an order of magnitude faster for generating very large samples

- samples = np.random.normal(size=(4, 4)); samples

```
array([[ 0.45978649,  0.08724787,  1.23489383, -2.41116764],
       [ 0.04595016,  0.92338218,  1.8235363 ,  0.66658714],
       [ 0.64525798,  0.9377648 , -0.44780425,  1.0675861 ],
       [-0.49855427, -0.05876717,  0.10930774,  0.57189039]])
```

Table 4-8. Partial list of numpy.random functions

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

Week 9-10:
Getting Started with Pandas

OUTLINE

Introduction to pandas Data Structures
Essential Functionality
Summarizing and Computing Descriptive Statistics
Handling Missing Data
Hierarchical Indexing
Other pandas Topics

INTRODUCTION TO PANDAS DATA STRUCTURES

Series

- A Series is a one-dimensional array-like object containing an array of data and an associated array of data labels, called its index
- import pandas as pd
- obj = pd.Series([4, 7, -5, 3]); obj
- The string representation of a Series shows the index on the left and the values on the right

```
0    4
1    7
2   -5
3    3
dtype: int64
```

Series

- You can get the array representation and index object of the Series via its **values** and **index** attributes
- obj.values
- Out[]: array([4, 7, -5, 3], dtype=int64)
- obj.index
- Out[]: RangeIndex(start=0, stop=4, step=1)

- A Series's index can be altered in place by assignment
- obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']; obj

```
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

Series

- It will be desirable to create a Series with an index identifying each data point
- obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c']); obj2

```
d     4
b     7
a    -5
c     3
dtype: int64
```

- Compared with a regular NumPy array, you can use values in the index when selecting single values or a set of values
- obj2['a']
- Out[]: -5
- obj2['d'] = 6
- obj2[['c', 'a', 'd']]

```
c     3
a    -5
d     6
dtype: int64
```

Series

- NumPy array operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link
- obj2[obj2 > 0]; obj2 * 2; np.exp(obj2)

```
d    6
b    7
c    3
dtype: int64
```

```
d    12
b    14
a   -10
c     6
dtype: int64
```

```
d      403.428793
b     1096.633158
a        0.006738
c       20.085537
dtype: float64
```

Series

- Should you have data contained in a Python dict, you can create a Series from it by passing the dict
- sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
- obj3 = pd.Series(sdata); obj3

```
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

Series

- When only passing a dict, the index in the resulting Series will have the dict's keys in sorted order
- states = ['California', 'Ohio', 'Oregon', 'Texas']
- obj4 = pd.Series(sdata, index=states); obj4
- NaN (not a number) is considered in pandas to mark missing or NA values

```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

Series

- The **isnull** and **notnull** functions in pandas should be used to detect missing data
- import pandas as pd
- pd.isnull(obj4); pd.notnull(obj4); obj3 + obj4;

```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

```
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

Series

- Both the Series object itself and its index have a name attribute, which integrates with other key areas of pandas functionality
- obj4.name = 'population'
- obj4.index.name = 'state'; obj4

```
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

DataFrame

- A DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.)
- There are numerous ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays
- data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'], 'year': [2000, 2001, 2002, 2001, 2002], 'pop': [1.5. 1.7, 3.6, 2.4, 2.9]}

DataFrame

- frame = pd.DataFrame(data); frame

```
   pop    state  year
0  1.5     Ohio  2000
1  1.7     Ohio  2001
2  3.6     Ohio  2002
3  2.4   Nevada  2001
4  2.9   Nevada  2002
```

- If you pass a column that isn't contained in data , it will appear with NA values in the result

- frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'], index=['one', 'two', 'three', 'four', 'five']); frame2

```
       year   state  pop  debt
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7   NaN
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4   NaN
five   2002  Nevada  2.9   NaN
```

DataFrame

- A column in a DataFrame can be retrieved either by dict-like notation or by attribute
- frame2['state']; frame2.year

```
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
Name: state, dtype: object
```

```
one      2000
two      2001
three    2002
four     2001
five     2002
Name: year, dtype: int64
```

DataFrame

- Rows can be retrieved by position or name by a couple of methods, such as the **loc** indexing field
- frame2.loc['three'];
  frame2.iloc[2]

```
year      2002
state     Ohio
pop        3.6
debt       NaN
Name: three, dtype: object
```

```
year      2002
state     Ohio
pop        3.6
debt       NaN
Name: three, dtype: object
```

- Columns can be modified by assignment
- frame2['debt'] = 16.5; frame2

```
       year   state  pop  debt
one    2000    Ohio  1.5  16.5
two    2001    Ohio  1.7  16.5
three  2002    Ohio  3.6  16.5
four   2001  Nevada  2.4  16.5
five   2002  Nevada  2.9  16.5
```

DataFrame

- When assigning lists or arrays to a column, the value's length must match the length of the DataFrame
- If you assign a Series, it will be instead conformed exactly to the DataFrame's index, inserting missing values in any holes

- val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
- frame2['debt'] = val; frame2

```
       year   state  pop  debt
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7  -1.2
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4  -1.5
five   2002  Nevada  2.9  -1.7
```

DataFrame

- Assigning a column that doesn't exist will create a new column
- frame2['eastern'] = frame2.state == 'Ohio'; frame2

```
       year   state  pop  debt  eastern
one    2000    Ohio  1.5   NaN     True
two    2001    Ohio  1.7  -1.2     True
three  2002    Ohio  3.6   NaN     True
four   2001  Nevada  2.4  -1.5    False
five   2002  Nevada  2.9  -1.7    False
```

- The **del** keyword will delete columns
- del frame2['eastern']
- frame2.columns
- Out[]: Index(['year', 'state', 'pop', 'debt'], dtype='object')

DataFrame

- Another common form of data is a nested dict of dicts format
- If passed to DataFrame, it will interpret the outer dict keys as the columns and the inner keys as the row indices
- The keys in the inner dicts are unioned and sorted to form the index in the result This isn't true if an explicit index is specified

- pop = {'Nevada': {2001: 2.4, 2002: 2.9}, 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
- frame3 = pd.DataFrame(pop); frame3

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

DataFrame

- Dicts of Series are treated in the same way
- pdata = {'Ohio2': frame3['Ohio'][:-1], 'Nevada2': frame3['Nevada'][:2]}
- DataFrame(pdata)

	Nevada2	Ohio2
2000	NaN	1.5
2001	2.4	1.7

- If a DataFrame's index and columns have their name attributes set, these will also be displayed
- frame3.index.name = 'year'; frame3.columns.name = 'state'; frame3

state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

Table 5-1. Possible data inputs to DataFrame constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the "dict of arrays" case
dict of Series	Each value becomes a column. Indexes from each Series are unioned together to form the result's row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are unioned to form the row index as in the "dict of Series" case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result

Index Objects

- Index objects are immutable
- obj = pd.Series(range(3), index=['a', 'b', 'c'])
- index = obj.index; index
- Out[]: Index(['a', 'b', 'c'], dtype='object')
- index[1] = 'd'
- Out[]: TypeError: Index does not support mutable operations
- In addition to being array-like, an Index also functions as a fixed-size set
- 'Ohio' in frame3.columns
- Out[]: True
- 2003 in frame3.index
- Out[]: False

ESSENTIAL FUNCTIONALITY

Reindexing

- Calling **reindex** on this Series rearranges the data according to the new index, introducing missing values if any index values were not already present
- obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c']); obj

```
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

- obj.reindex(['a', 'b', 'c', 'd', 'e']); obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)

```
a   -5.3          a   -5.3
b    7.2          b    7.2
c    3.6          c    3.6
d    4.5          d    4.5
e    NaN          e    0.0
dtype: float64 dtype: float64
```

Reindexing

- For ordered data like time series, it may be desirable to do some interpolation or filling of values when reindexing
- The method option allows us to do this, using a method such as **ffill** which forward fills the values
- obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4]); obj3

```
0       blue
2     purple
4     yellow
dtype: object
```

- obj3.reindex(range(6), method='ffill')

```
0       blue
1       blue
2     purple
3     purple
4     yellow
5     yellow
dtype: object
```

Table 5-4. reindex method (interpolation) options

Argument	Description
`ffill` or `pad`	Fill (or carry) values forward
`bfill` or `backfill`	Fill (or carry) values backward

Reindexing

- With DataFrame, reindex can alter either the (row) index, columns, or both
- frame = pd.DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'], columns=['Ohio', 'Texas', 'California'])
  frame
- states = ['Texas', 'Utah', 'California']
- [illegible]

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

	Texas	Utah	California
a	1.0	NaN	2.0
b	NaN	NaN	NaN
c	4.0	NaN	5.0
d	7.0	NaN	8.0

Dropping entries from an axis

- The **drop** method will return a new object with the indicated value or values deleted from an axis
- data = pd.DataFrame(np.arange(16).reshape((4, 4)), index=['Ohio', 'Colorado', 'Utah', 'New York'], columns=['one', 'two', 'three', 'four']); data
- data.drop(['Colorado', 'Ohio']); data.drop(['two', 'four'], axis=1)

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

Indexing, selection, and filtering

- Series indexing works analogously to NumPy array indexing, except you can use the Series's index values instead of only integers
- obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd']); obj
- obj[2:4]; obj[['b', 'a', 'd']]; obj[obj < 2]

```
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

```
c    2.0
d    3.0
dtype: float64
```

```
b    1.0
a    0.0
d    3.0
dtype: float64
```

```
a    0.0
b    1.0
dtype: float64
```

Indexing, selection, and filtering

- Slicing with labels behaves differently than normal Python slicing in that the endpoint is inclusive
- obj['b':'c'] = 5

```
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

- Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence
- data = pd.DataFrame(np.arange(16).reshape((4, 4)), index=['Ohio', 'Colorado', 'Utah', 'New York'], columns=['one', 'two', 'three', 'four']); data
- data['two']; data[data['three'] > 5]

```
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

```
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int32
```

```
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Indexing, selection, and filtering

- Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison
- data < 5; data[data < 5] = 0; data

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

- data.iloc[[1, 2], [3, 0, 1]];
  data.iloc[:, :3][data.three > 5]

	four	one	two
Colorado	7	0	5
Utah	11	8	9

	one	two	three
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

Table 5-6. Indexing options with DataFrame

Type	Notes
`obj[val]`	Select single column or sequence of columns from the DataFrame. Special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion).
`obj.ix[val]`	Selects single row of subset of rows from the DataFrame.
`obj.ix[:, val]`	Selects single column of subset of columns.
`obj.ix[val1, val2]`	Select both rows and columns.
`reindex` method	Conform one or more axes to new indexes.
`xs` method	Select single row or column as a Series by label.
`icol, irow` methods	Select single column or row, respectively, as a Series by integer location.
`get_value, set_value` methods	Select single value by row and column label.

Arithmetic and data alignment

- One of the most important pandas features is the behavior of arithmetic between objects with different indexes
- s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e']); s1
- s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g']); s2
- s1 + s2

```
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
```

```
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

Arithmetic and data alignment

- df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'), index=['Ohio', 'Texas', 'Colorado']); df1
- df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'), index=['Utah', 'Ohio', 'Texas', 'Oregon']);df2
- df1+df2

```
            b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0
```

```
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

```
            b   c     d   e
Colorado  NaN NaN   NaN NaN
Ohio      3.0 NaN   6.0 NaN
Oregon    NaN NaN   NaN NaN
Texas     9.0 NaN  12.0 NaN
Utah      NaN NaN   NaN NaN
```

Arithmetic methods with fill values

- In arithmetic operations between differently-indexed objects, you might want to fill with a special value, like 0
- df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd')); df1
- df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde')); df2
- df1 + df2; df1.add(df2, fill_value=0)

```
     a    b     c     d
0  0.0  1.0   2.0   3.0
1  4.0  5.0   6.0   7.0
2  8.0  9.0  10.0  11.0
      a     b     c     d     e
0   0.0   1.0   2.0   3.0   4.0
1   5.0   6.0   7.0   8.0   9.0
2  10.0  11.0  12.0  13.0  14.0
3  15.0  16.0  17.0  18.0  19.0
       a      b      c      d   e
0    0.0    2.0    4.0    6.0 NaN
1    9.0   11.0   13.0   15.0 NaN
2   18.0   20.0   22.0   24.0 NaN
3    NaN    NaN    NaN    NaN NaN
       a      b      c      d     e
0    0.0    2.0    4.0    6.0   4.0
1    9.0   11.0   13.0   15.0   9.0
2   18.0   20.0   22.0   24.0  14.0
3   15.0   16.0   17.0   18.0  19.0
```

Arithmetic methods with fill values

- When reindexing a Series or DataFrame, you can also specify a different fill value
- df1.reindex(columns=df2.columns, fill_value=0)

```
     a    b     c     d  e
0  0.0  1.0   2.0   3.0  0
1  4.0  5.0   6.0   7.0  0
2  8.0  9.0  10.0  11.0  0
```

Table 5-7. Flexible arithmetic methods

Method	Description
add	Method for addition (+)
sub	Method for subtraction (-)
div	Method for division (/)
mul	Method for multiplication (*)

Operations between DataFrame and Series

- arr = np.arange(12.).reshape((3, 4)); arr
- arr[0]
- Out[]: array([0., 1., 2., 3.])
- arr - arr[0]

```
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.]])

array([[ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 8.,  8.,  8.,  8.]])
```

- frame = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'), index=['Utah', 'Ohio', 'Texas', 'Oregon']); frame
- frame.iloc[0]; series

```
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

Operations between DataFrame and Series

- By default, arithmetic between DataFrame and Series matches the index of the Series on the DataFrame's columns, broadcasting down the rows
- frame - series

```
          b    d    e
Utah    0.0  0.0  0.0
Ohio    3.0  3.0  3.0
Texas   6.0  6.0  6.0
Oregon  9.0  9.0  9.0
```

- If an index value is not found in either the DataFrame's columns or the Series's index, the objects will be reindexed to form the union:
- series2 = pd.Series(range(3), index=['b', 'e', 'f']); frame + series2

```
b    0
e    1
f    2
dtype: int32
```

```
          b   d     e   f
Utah    0.0 NaN   3.0 NaN
Ohio    3.0 NaN   6.0 NaN
Texas   6.0 NaN   9.0 NaN
Oregon  9.0 NaN  12.0 NaN
```

Operations between DataFrame and Series

- If you want to instead broadcast over the columns, matching on the rows, you have to use one of the arithmetic methods
- series3 = frame['d']; series3
- frame.sub(series3, axis=0)

```
Utah      1.0
Ohio      4.0
Texas     7.0
Oregon    10.0
Name: d, dtype: float64
```

```
          b    d    e
Utah   -1.0  0.0  1.0
Ohio   -1.0  0.0  1.0
Texas  -1.0  0.0  1.0
Oregon -1.0  0.0  1.0
```

Function application and mapping

- NumPy ufuncs work fine with pandas objects
- Another frequent operation is applying a function on 1D arrays to each column or row
- frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'), index=['Utah', 'Ohio', 'Texas', 'Oregon']); frame
- f = lambda x: x.max() - x.min()
- frame.apply(f); frame.apply(f, axis=1)

```
                b         d         e
Utah    -1.793630  2.504958 -2.057384
Ohio     0.422265 -0.320772 -0.531460
Texas    0.247096  0.517801  0.203822
Oregon   0.442154  0.365938 -0.140527
```

```
b    2.235784
d    2.825730
e    2.261206
dtype: float64
```

```
Utah      4.562342
Ohio      0.953725
Texas     0.313979
Oregon    0.582681
dtype: float64
```

Function application and mapping

- The function passed to **apply** need not return a scalar value, it can also return a Series with multiple values
- def f(x):
  - return Series([x.min(), x.max()], index=['min', 'max'])
- frame.apply(f)

```
            b         d         e
min -1.793630 -0.320772 -2.057384
max  0.442154  2.504958  0.203822
```

- Element-wise Python functions can be used, tooformat = lambda x: '%.2f' % x
- frame.applymap(format)

```
              b     d     e
Utah      -1.79  2.50 -2.06
Ohio       0.42 -0.32 -0.53
Texas      0.25  0.52  0.20
Oregon     0.44  0.37 -0.14
```

Sorting and ranking

- To sort lexicographically by row or column index, use the sort_index methodt:
- obj = pd.Series(range(4), index=['d', 'a', 'b', 'c']); obj
- obj.sort_index()

```
d    0          a    1
a    1          b    2
b    2          c    3
c    3          d    0
dtype: int64    dtype: int64
```

- With a DataFrame, you can sort by index on either axis
- frame = pd.DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'], columns=['d', 'a', 'b', 'c']); frame
- frame.sort_index(axis=1)

```
       d  a  b  c          a  b  c  d
three  0  1  2  3  three   1  2  3  0
one    4  5  6  7  one     5  6  7  4
```

Sorting and ranking

- The data is sorted in ascending order by default, but can be sorted in descending order, too
- frame.sort_index(axis=1, ascending=False)

```
       d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

- To sort a Series by its values, use its **sort_values** method

- Any missing values are sorted to the end of the Series by default
- obj = pd.Series([4, 7, -3, 2]); obj.sort_values ()
- obj = pd.Series([4, np.nan, 7, np.nan, -3, 2]); obj. sort_values()

```
2   -3
3    2
0    4
1    7
dtype: int64
```

```
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

Sorting and ranking

- On DataFrame, you may want to sort by the values in one or more columns. To do so, pass one or more column names to the **by** option
- frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

	a	b
0	0	4
1	1	7
2	0	-3
3	1	2

- frame.sort_values(by='b')

	a	b
2	0	-3
3	1	2
0	0	4
1	1	7

- frame.sort_values(by=['a', 'b'])

	a	b
2	0	-3
0	0	4
3	1	2
1	1	7

Sorting and ranking

- Ranking is closely related to sorting, assigning ranks from one through the number of valid data points in an array
- The **rank** methods for Series and DataFrame are the place to look; by default rank breaks ties by assigning each group the mean rank

- Ranks can also be assigned according to the order they're observed in the data
- obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
- obj.rank();

```
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

```
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

Sorting and ranking

- DataFrame can compute ranks over the rows or the columns
- frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1], 'c': [-2, 5, 8, -2.5]}); frame
- frame.rank(axis=1)

Table 5-8. Tie-breaking methods with rank

Method	Description
'average'	Default: assign the average rank to each entry in the equal group.
'min'	Use the minimum rank for the whole group.
'max'	Use the maximum rank for the whole group.
'first'	Assign ranks in the order the values appear in the data.

```
   a    b    c
0  0  4.3 -2.0
1  1  7.0  5.0
2  0 -3.0  8.0
3  1  2.0 -2.5
```

```
     a    b    c
0  2.0  3.0  1.0
1  1.0  3.0  2.0
2  2.0  1.0  3.0
3  2.0  3.0  1.0
```

Axis indexes with duplicate values

- While many pandas functions (like reindex) require that the labels be unique, it's not mandatory
- obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c']); obj
- obj.index.is_unique
- Out[]: False
- obj['a']

```
a    0
a    1
b    2
b    3
c    4
dtype: int32
```

```
a    0
a    1
dtype: int32
```

- The same logic extends to indexing rows in a DataFrame
- df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b']); df
-

```
          0         1         2
a -0.939395  0.639066  0.467387
a  0.263369 -1.709998 -0.312073
b -1.393720  0.107396  0.194525
b  0.250044  1.716324 -0.378712
```

```
          0         1         2
b -1.393720  0.107396  0.194525
b  0.250044  1.716324 -0.378712
```

SUMMARIZING AND COMPUTING DESCRIPTIVE STATISTICS

Summarizing and Computing Descriptive Statistics

- pandas objects are equipped with a set of common mathematical and statistical methods
- Most of these fall into the category of reductions or summary statistics, methods that extract a single from a Series or a Series of values from the rows or columns of a DataFrame
- Compared with the equivalent methods of vanilla NumPy arrays, they are all built from the ground up to exclude missing data

Summarizing and Computing Descriptive Statistics

- df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5], [np.nan, np.nan], [0.75, -1.3]], index=['a', 'b', 'c', 'd'], columns=['one', 'two']); df
- df.sum(); df.sum(axis=1)

```
   one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3
```

```
one     9.25
two    -5.80
dtype: float64
```

```
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

- NA values are excluded unless the entire slice (row or column in this case) is NA. This can be disabled using the **skipna** option
- df.mean(axis=1, skipna=False)

```
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

Summarizing and Computing Descriptive Statistics

- Some methods, like **idxmin** and **idxmax**, return indirect statistics like the index value where the minimum or maximum values are attained. Other methods are accumulations
- df.idxmax(); df.cumsum()

```
one    b
two    d
dtype: object
```

```
   one  two
a  1.40  NaN
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8
```

- **describe** is one such example, producing multiple summary statistics in one shot
- df.describe()
- obj = pd.Series(['a', 'a', 'b', 'c'] * 4); obj.describe()

```
            one       two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

```
count      16
unique      3
top         a
freq        8
dtype: object
```

Table 5-10. Descriptive and summary statistics

Method	Description
`count`	Number of non-NA values
`describe`	Compute set of summary statistics for Series or each DataFrame column
`min, max`	Compute minimum and maximum values
`argmin, argmax`	Compute index locations (integers) at which minimum or maximum value obtained, respectively
`idxmin, idxmax`	Compute index values at which minimum or maximum value obtained, respectively
`quantile`	Compute sample quantile ranging from 0 to 1
`sum`	Sum of values
`mean`	Mean of values
`median`	Arithmetic median (50% quantile) of values
`mad`	Mean absolute deviation from mean value
`var`	Sample variance of values
`std`	Sample standard deviation of values
`skew`	Sample skewness (3rd moment) of values
`kurt`	Sample kurtosis (4th moment) of values
`cumsum`	Cumulative sum of values
`cummin, cummax`	Cumulative minimum or maximum of values, respectively
`cumprod`	Cumulative product of values
`diff`	Compute 1st arithmetic difference (useful for time series)
`pct_change`	Compute percent changes

Correlation and Covariance

- Let's consider some DataFrames of stock prices and volumes obtained from Yahoo! Finance
- from pandas_datareader import data
- all_data = {}
- for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']:
  - all_data[ticker] = data.get_data_yahoo(ticker, '10/1/2017', '10/10/2017')
- price = pd.DataFrame({tic: data['Adj Close']
  - for tic, data in all_data.items()})
- volume = pd.DataFrame({tic: data['Volume']
  - for tic, data in all_data.items()})
- returns = price.pct_change()

get_data_yahoo method

- AAPL=data.get_data_yahoo('AAPL', '10/1/2017', '10/10/2017')
- type(AAPL)
- Out[]: pandas.core.frame.DataFrame
- AAPL; AAPL['Adj Close']

```
                 Open        High         Low       Close   Adj Close  \
Date
2017-10-02  154.259995  154.449997  152.720001  153.809998  153.809998
2017-10-03  154.009995  155.089996  153.910004  154.479996  154.479996
2017-10-04  153.630005  153.860001  152.460007  153.479996  153.479996
2017-10-05  154.179993  155.440002  154.050003  155.389999  155.389999
2017-10-06  154.970001  155.490005  154.559998  155.300003  155.300003
2017-10-09  155.809998  156.729996  155.490005  155.839996  155.839996
2017-10-10  156.059998  158.000000  155.100006  155.899994  155.899994

               Volume
Date
2017-10-02  18698800
2017-10-03  16230300
2017-10-04  20163800
2017-10-05  21283800
2017-10-06  17407600
2017-10-09  16262900
2017-10-10  15617000
```

```
Date
2017-10-02    153.809998
2017-10-03    154.479996
2017-10-04    153.479996
2017-10-05    155.389999
2017-10-06    155.300003
2017-10-09    155.839996
2017-10-10    155.899994
Name: Adj Close, dtype: float64
```

get_data_yahoo method

- type(all_data)
- Out[]: dict
- all_data.keys()
- Out[]: dict_keys(['AAPL', 'IBM', 'MSFT', 'GOOG'])
- type(price)
- Out[]: pandas.core.frame.DataFrame
- price; volume; returns

Date	AAPL	GOOG	IBM	MSFT
2017-10-02	153.809998	953.270020	146.660004	74.610001
2017-10-03	154.479996	957.789978	146.779999	74.260002
2017-10-04	153.479996	951.679993	146.479996	74.690002
2017-10-05	155.389999	969.960022	146.720001	75.970001
2017-10-06	155.300003	978.890015	146.479996	76.000000
2017-10-09	155.839996	977.000000	147.389999	76.290001
2017-10-10	155.899994	972.599976	148.500000	76.290001

Date	AAPL	GOOG	IBM	MSFT
2017-10-02	18698800	1283400	2973200	15304800
2017-10-03	16230300	888300	2302700	12190400
2017-10-04	20163800	952400	2244400	13317700
2017-10-05	21283800	1213800	2686500	21195300
2017-10-06	17407600	1173900	2623200	13959800
2017-10-09	16262900	891400	2682600	11386500
2017-10-10	15617000	968400	4032600	13944500

Date	AAPL	GOOG	IBM	MSFT
2017-10-02	NaN	NaN	NaN	NaN
2017-10-03	0.004356	0.004742	0.000818	-0.004691
2017-10-04	-0.006473	-0.006379	-0.002044	0.005790
2017-10-05	0.012445	0.019208	0.001638	0.017137
2017-10-06	-0.000579	0.009207	-0.001636	0.000395
2017-10-09	0.003477	-0.001931	0.006212	0.003816
2017-10-10	0.000385	-0.004504	0.007531	0.000000

Correlation and Covariance

- returns.MSFT.corr(returns.IBM)
- Out[]: -0.07630883561389795 2
- returns.MSFT.cov(returns.IBM)
- Out[]: -2.2753216233070484e-06
- returns.corr(); returns.cov()
- returns.corrwith(returns.IBM); returns.corrwith(volume)

	AAPL	GOOG	IBM	MSFT
AAPL	1.000000	0.789706	0.273413	0.499639
GOOG	0.789706	1.000000	-0.284305	0.523151
IBM	0.273413	-0.284305	1.000000	-0.076309
MSFT	0.499639	0.523151	-0.076309	1.000000

	AAPL	GOOG	IBM	MSFT
AAPL	0.000039	0.000048	0.000007	0.000023
GOOG	0.000048	0.000094	-0.000011	0.000038
IBM	0.000007	-0.000011	0.000016	-0.000002
MSFT	0.000023	0.000038	-0.000002	0.000056

```
AAPL     0.273413 AAPL     0.202547
GOOG    -0.284305 GOOG     0.805371
IBM      1.000000 IBM      0.759415
MSFT    -0.076309 MSFT     0.844384
dtype: float64   dtype: float64
```

Unique Values, Value Counts, and Membership

- The first function is **unique**, which gives you an array of the unique values in a Series
- **value_counts** computes a Series containing value frequencies
- **isin** performs vectorized set membership and can be very useful in filtering a data set down to a subset of values in a Series or column in a DataFrame
- obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c']); obj
- uniques = obj.unique(); uniques
- Out[]: array(['c', 'a', 'd', 'b'], dtype=object)
- obj.value_counts()
- mask = obj.isin(['b', 'c']); mask

```
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object

a    3
c    3
b    2
d    1
dtype: int64
```

```
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
```

OTHER PANDAS TOPICS

Integer Indexing

- We have an index containing 0, 1, 2, but inferring what the user wants (label-based indexing or position-based) is difficult. On the other hand, with a non-integer index, there is no potential for ambiguity

```
0    0.0          a    0.0
1    1.0          b    1.0
2    2.0          c    2.0
dtype: float64  dtype: float64
```

- ser = pd.Series(np.arange(3.))
- ser[-1] #위치 색인 불가능. 이름 색인만 가능
- ser[2]
- Out[]: 2.0
- ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c']); ser2
- ser2[-1] #위치 색인 가능
- Out[]: 2.0

Integer Indexing

- ser3 = pd.Series(range(3), index=[-5, 1, 3]); ser3
- ser3.iloc[2]; ser3.loc[3]
- Out[]: 2
- frame = pd.DataFrame(np.arange(6).reshape(3, 2), index=[2, 0, 1])
- frame.iloc[0]; frame.iloc[:,0]

```
-5    0
 1    1
 3    2
dtype: int64
```

```
   0  1
2  0  1
0  2  3
1  4  5
```

```
0    1
1    3
Name: 2, dtype: int64
```

```
2    0
0    2
1    4
Name: 0, dtype: int64
```

Panel Data

- To create a Panel, you can use a dict of DataFrame objects or a three-dimensional ndarray
- from pandas_datareader import data
- pdata = pd.Panel(dict((stk, data.get_data_yahoo(stk, '10/1/2017', '10/7/2017')) for stk in ['AAPL', 'IBM', 'MSFT', 'GOOG'])); pdata

```
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 5 (major_axis) x 6 (minor_axis)
Items axis: AAPL to MSFT
Major_axis axis: 2017-10-02 00:00:00 to 2017-10-06 00:00:00
Minor_axis axis: Open to Volume
```

Panel Data

- pdata2 = pdata.swapaxes('items', 'minor')

```
__main__:1: DeprecationWarning:
Panel is deprecated and will be removed in a future version.
```

- pdata2['Adj Close']; pdata2.loc[:, '10/2/2017', :]; pdata2.loc[:, :, 'AAPL']

Date	AAPL	GOOG	IBM	MSFT
2017-10-02	153.809998	953.270020	146.660004	74.610001
2017-10-03	154.479996	957.789978	146.779999	74.260002
2017-10-04	153.479996	951.679993	146.479996	74.690002
2017-10-05	155.389999	969.960022	146.720001	75.970001
2017-10-06	155.300003	978.890015	146.479996	76.000000

	Open	High	Low	Close	Adj Close	Volume
AAPL	154.259995	154.449997	152.720001	153.809998	153.809998	18698800.0
GOOG	959.979980	962.539978	947.840027	953.270020	953.270020	1283400.0
IBM	145.350006	146.869995	145.210007	146.660004	146.660004	2973200.0
MSFT	74.709999	75.010002	74.300003	74.610001	74.610001	15304800.0

Date	Open	High	Low	Close	Adj Close
2017-10-02	154.259995	154.449997	152.720001	153.809998	153.809998
2017-10-03	154.009995	155.089996	153.910004	154.479996	154.479996
2017-10-04	153.630005	153.860001	152.460007	153.479996	153.479996
2017-10-05	154.179993	155.440002	154.050003	155.389999	155.389999
2017-10-06	154.970001	155.490005	154.559998	155.300003	155.300003

Date	Volume
2017-10-02	18698800.0
2017-10-03	16230300.0
2017-10-04	20163800.0
2017-10-05	21283800.0
2017-10-06	17407600.0

Panel Data

- pdata2.loc['Adj Close', '10/4/2017':, :]
- stacked = pdata2.loc[:, '10/4/2017':, :].to_frame(); stacked
- stacked.to_panel()

Date	AAPL	GOOG	IBM	MSFT
2017-10-04	153.479996	951.679993	146.479996	74.690002
2017-10-05	155.389999	969.960022	146.720001	75.970001
2017-10-06	155.300003	978.890015	146.479996	76.000000

Date	minor	Open	High	Low	Close	Adj Close
2017-10-04	AAPL	153.630005	153.860001	152.460007	153.479996	153.479996
	GOOG	957.000000	960.390015	950.690002	951.679993	951.679993
	IBM	147.000000	147.020004	146.110001	146.479996	146.479996
	MSFT	74.089996	74.720001	73.709999	74.690002	74.690002
2017-10-05	AAPL	154.179993	155.440002	154.050003	155.389999	155.389999
	GOOG	955.489990	970.909973	955.179993	969.960022	969.960022
	IBM	146.679993	147.539993	146.479996	146.720001	146.720001
	MSFT	75.220001	76.120003	74.959999	75.970001	75.970001
2017-10-06	AAPL	154.970001	155.490005	154.559998	155.300003	155.300003
	GOOG	966.700012	979.460022	963.359985	978.890015	978.890015
	IBM	146.639999	146.850006	146.320007	146.479996	146.479996
	MSFT	75.669998	76.029999	75.540001	76.000000	76.000000

```
<class 'pandas.core.panel.Panel'>
Dimensions: 6 (items) x 3 (major_axis) x 4 (minor_axis)
Items axis: Open to Volume
Major_axis axis: 2017-10-04 00:00:00 to 2017-10-06 00:00:00
Minor_axis axis: AAPL to MSFT
```

Panel data

- Panel is a somewhat less-used, but still important container for 3-dimensional data. The term panel data is derived from econometrics and is partially responsible for the name pandas: pan(el)-da(ta)-s
- Panel is deprecated and will be removed in a future version
- **items**: axis 0, each item corresponds to a DataFrame contained inside
- **major_axis**: axis 1, it is the **index** (rows) of each of the DataFrames
- **minor_axis**: axis 2, it is the **columns** of each of the DataFrames

pdata

- pdata.items
- Out[]: Index(['AAPL', 'GOOG', 'IBM', 'MSFT'], dtype='object')
- pdata.major_axis

```
DatetimeIndex(['2017-10-02', '2017-10-03', '2017-10-04', '2017-10-05',
               '2017-10-06'],
```

- pdata.minor_axis
- Out[]: Index(['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume'], dtype='object')

pdata2

- pdata2=pdata.transpose(2,1,0); pdata2
- pdata2['Adj Close']
- pdata2.major_xs('10/2/2017')
- pdata2.minor_xs('Adj Close')

```
<class 'pandas.core.panel.Panel'>
Dimensions: 6 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: Open to Volume
Major_axis axis: 2017-10-02 00:00:00 to 2017-10-06 00:00:00
Minor_axis axis: AAPL to MSFT
```

Date	AAPL	GOOG	IBM	MSFT
2017-10-02	153.809998	953.270020	146.660004	74.610001
2017-10-03	154.479996	957.789978	146.779999	74.260002
2017-10-04	153.479996	951.679993	146.479996	74.690002
2017-10-05	155.389999	969.960022	146.720001	75.970001
2017-10-06	155.300003	978.890015	146.479996	76.000000

Date	Open	High	Low	Close	Adj Close
2017-10-02	154.259995	154.449997	152.720001	153.809998	153.809998
2017-10-03	154.009995	155.089996	153.910004	154.479996	154.479996
2017-10-04	153.630005	153.860001	152.460007	153.479996	153.479996
2017-10-05	154.179993	155.440002	154.050003	155.389999	155.389999
2017-10-06	154.970001	155.490005	154.559998	155.300003	155.300003

	Open	High	Low	Close	Adj Close	Volume
AAPL	154.259995	154.449997	152.720001	153.809998	153.809998	18698800.0
GOOG	959.979980	962.539978	947.840027	953.270020	953.270020	1283400.0
IBM	145.350006	146.869995	145.210007	146.660004	146.660004	2973200.0
MSFT	74.709999	75.010002	74.300003	74.610001	74.610001	15304800.0

Date	Volume
2017-10-02	18698800.0
2017-10-03	16230300.0
2017-10-04	20163800.0
2017-10-05	21283800.0
2017-10-06	17407600.0

to_frame method

- Transform wide format into long (stacked) format as DataFrame whose columns are the Panel's items and whose index is a MultiIndex formed of the Panel's major and minor axes
- pdata.to_frame()

		AAPL	GOOG	IBM	MSFT
Date	minor				
2017-10-02	Open	1.542600e+02	9.599800e+02	1.453500e+02	7.471000e+01
	High	1.544500e+02	9.625400e+02	1.468700e+02	7.501000e+01
	Low	1.527200e+02	9.478400e+02	1.452100e+02	7.430000e+01
	Close	1.538100e+02	9.532700e+02	1.466600e+02	7.461000e+01
	Adj Close	1.538100e+02	9.532700e+02	1.466600e+02	7.461000e+01
	Volume	1.869880e+07	1.283400e+06	2.973200e+06	1.530480e+07
2017-10-03	Open	1.540100e+02	9.540000e+02	1.466900e+02	7.467000e+01
	High	1.550900e+02	9.580000e+02	1.472000e+02	7.488000e+01
	Low	1.539100e+02	9.491400e+02	1.463400e+02	7.419000e+01
	Close	1.544800e+02	9.577900e+02	1.467800e+02	7.426000e+01
	Adj Close	1.544800e+02	9.577900e+02	1.467800e+02	7.426000e+01
	Volume	1.623030e+07	8.883000e+05	2.302700e+06	1.219040e+07
2017-10-04	Open	1.536300e+02	9.570000e+02	1.470000e+02	7.409000e+01
	High	1.538600e+02	9.603900e+02	1.470200e+02	7.472000e+01
	Low	1.524600e+02	9.506900e+02	1.461100e+02	7.371000e+01
	Close	1.534800e+02	9.516800e+02	1.464800e+02	7.469000e+01
	Adj Close	1.534800e+02	9.516800e+02	1.464800e+02	7.469000e+01
	Volume	2.016380e+07	9.524000e+05	2.244400e+06	1.331770e+07
2017-10-05	Open	1.541800e+02	9.554900e+02	1.466800e+02	7.522000e+01
	High	1.554400e+02	9.709100e+02	1.475400e+02	7.612000e+01
	Low	1.540500e+02	9.551800e+02	1.464800e+02	7.496000e+01
	Close	1.553900e+02	9.699600e+02	1.467200e+02	7.597000e+01
	Adj Close	1.553900e+02	9.699600e+02	1.467200e+02	7.597000e+01
	Volume	2.128380e+07	1.213800e+06	2.686500e+06	2.119530e+07
2017-10-06	Open	1.549700e+02	9.667000e+02	1.466400e+02	7.567000e+01
	High	1.554900e+02	9.794600e+02	1.468500e+02	7.603000e+01
	Low	1.545600e+02	9.633600e+02	1.463200e+02	7.554000e+01
	Close	1.553000e+02	9.788900e+02	1.464800e+02	7.600000e+01
	Adj Close	1.553000e+02	9.788900e+02	1.464800e+02	7.600000e+01
	Volume	1.740760e+07	1.173900e+06	2.623200e+06	1.395980e+07

Week 11:
Data Cleaning and Preparation

Handling Missing Data

Data Transformation

String Manipulation

OUTLINE

HANDLING MISSING DATA

Handling Missing Data

- For numeric data, pandas uses the floating point value **NaN** (Not a Number) to represent missing data
- string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado']);

```
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object
```

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

- The built-in Python **None** value is also treated as **NA**(Not available) in object arrays
- string_data[0] = None
- string_data.isnull()

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

Handling Missing Data

Table 5-12. NA handling methods

Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`.
`isnull`	Return like-type object containing boolean values indicating which values are missing / NA.
`notnull`	Negation of `isnull`.

Filtering Out Missing Data

- While you have the option to filter out missing data by hand using pandas.**isnull** and boolean indexing, **dropna** can be helpful
- from numpy import nan as NA
- data = pd.Series([1, NA, 3.5, NA, 7]); data
- data.dropna(); data[data.notnull()]

```
0    1.0
1    NaN
2    3.5
3    NaN
4    7.0
dtype: float64
```

```
0    1.0
2    3.5
4    7.0
dtype: float64
```

Filtering Out Missing Data

- With DataFrame objects, **dropna** by default drops any row containing a missing value
- data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA], [NA, NA, NA], [NA, 6.5, 3.]]); data
- cleaned = data.dropna(); cleaned

```
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

```
     0    1    2
0  1.0  6.5  3.0
```

Filtering Out Missing Data

- Passing how='all' will only drop rows that are all NA
- data.dropna(how='all')
- data[4] = NA; data
- data.dropna(axis=1, how='all')

```
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0
```

```
     0    1    2   4
0  1.0  6.5  3.0 NaN
1  1.0  NaN  NaN NaN
2  NaN  NaN  NaN NaN
3  NaN  6.5  3.0 NaN
```

```
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

Filtering Out Missing Data

- Suppose you want to keep only rows containing a certain number of observations. You can indicate this with the **thresh** argument
- df = pd.DataFrame(np.random.randn(7, 3)); df
- df.loc[:4, 1] = NA; df.loc[:2, 2] = NA; df
- df.dropna(thresh=3)

```
          0         1         2
0  0.102481  0.106342  0.291524
1  1.756686  1.587244  1.291618
2  1.180782  1.309581 -0.673739
3 -0.851258  0.018325  0.302447
4  1.431770  0.504683 -0.866383
5  0.587037 -2.338839 -0.077326
6  1.559964  0.444927 -0.068242
```

```
          0         1         2
0  0.102481       NaN       NaN
1  1.756686       NaN       NaN
2  1.180782       NaN       NaN
3 -0.851258       NaN  0.302447
4  1.431770       NaN -0.866383
5  0.587037 -2.338839 -0.077326
6  1.559964  0.444927 -0.068242
```

```
          0         1         2
5 -0.650572  0.365794 -2.210859
6  1.687898 -0.262608  0.932170
```

Filling in Missing Data

- Rather than filtering out missing data (and potentially discarding other data along with it), you may want to fill in the "holes" in any number of ways. The **fillna** method is the workhorse function to use
- df.fillna(0); df.fillna({1: 0.5, 2: -1})

	0	1	2
0	-0.577087	0.000000	0.000000
1	0.523772	0.000000	0.000000
2	-0.713544	0.000000	0.000000
3	-1.860761	0.000000	0.560145
4	-1.265934	0.000000	-1.063512
5	0.332883	-2.359419	-0.199543
6	-1.541996	-0.970736	-1.307030

	0	1	2
0	-0.644354	0.500000	-1.000000
1	0.145960	0.500000	-1.000000
2	-0.429010	0.500000	-1.000000
3	-0.757190	0.500000	0.277761
4	-1.815087	0.500000	0.290200
5	-0.650572	0.365794	-2.210859
6	1.687898	-0.262608	0.932170

Filling in Missing Data

- The same interpolation methods available for reindexing can be used with **fillna**
- df = pd.DataFrame(np.random.randn(6, 3))
- df.iloc[2:, 1] = NA; df.iloc[4:, 2] = NA; df
- df.fillna(method='ffill', limit=2)
- You might pass the mean or median value of a Series
- data = pd.Series([1., NA, 3.5, NA, 7]); data
- data.fillna(data.mean())

```
          0         1         2
0 -1.144871  0.953442  1.339559
1  0.589104  1.441956  1.682556
2  0.065279       NaN  0.272161
3  1.044772       NaN  0.127885
4 -2.104738       NaN       NaN
5 -0.259037       NaN       NaN
```

```
          0         1         2
0 -1.144871  0.953442  1.339559
1  0.589104  1.441956  1.682556
2  0.065279  1.441956  0.272161
3  1.044772  1.441956  0.127885
4 -2.104738       NaN  0.127885
5 -0.259037       NaN  0.127885
```

```
0    1.0
1    NaN
2    3.5
3    NaN
4    7.0
dtype: float64
```

```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

Filling in Missing Data

Table 5-13. fillna function arguments

Argument	Description
`value`	Scalar value or dict-like object to use to fill missing values
`method`	Interpolation, by default `'ffill'` if function called with no other arguments
`axis`	Axis to fill on, default `axis=0`
`inplace`	Modify the calling object without producing a copy
`limit`	For forward and backward filling, maximum number of consecutive periods to fill

DATA TRANSFORMATION

Removing Duplicates

- data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'], 'k2': [1, 1, 2, 3, 3, 4, 4]}); data
- data.duplicated()
- data.drop_duplicates()
- data['v1'] = range(7); data
- data.drop_duplicates(['k1'])
- data.drop_duplicates(['k1', 'k2'], keep='last')

```
    k1   k2
0  one    1
1  two    1
2  one    2
3  two    3
4  one    3
5  two    4
6  two    4
```

```
0    False
1    False
2    False
3    False
4    False
5     True
6    False
dtype: bool
```

```
    k1   k2
0  one    1
1  two    1
2  one    2
3  two    3
4  one    3
5  two    4
```

```
    k1   k2  v1
0  one    1   0
1  two    1   1
2  one    2   2
3  two    3   3
4  one    3   4
5  two    4   5
6  two    4   6
```

```
    k1   k2  v1
0  one    1   0
1  two    1   1
```

```
    k1   k2  v1
0  one    1   0
1  two    1   1
2  one    2   2
3  two    3   3
4  one    3   4
6  two    4   6
```

Transforming Data Using a Function or Mapping

- data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon', 'Pastrami', 'corned beef', 'Bacon', 'pastrami', 'honey ham', 'nova lox'], 'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]}); data
- meat_to_animal = {'bacon': 'pig', 'pulled pork': 'pig', 'pastrami': 'cow', 'corned beef': 'cow', 'honey ham': 'pig', 'nova lox': 'salmon'}
- lowercased = data['food'].str.lower(); lowercased
- lowercased.map(meat_to_animal)
- data['animal'] = lowercased.map(meat_to_animal); data

```
          food  ounces
0        bacon     4.0
1  pulled pork     3.0
2        bacon    12.0
3     Pastrami     6.0
4  corned beef     7.5
5        Bacon     8.0
6     pastrami     3.0
7    honey ham     5.0
8     nova lox     6.0
```

```
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object
```

```
0       pig
1       pig
2       pig
3       cow
4       cow
5       pig
6       cow
7       pig
8    salmon
Name: food, dtype: object
```

```
          food  ounces  animal
0        bacon     4.0     pig
1  pulled pork     3.0     pig
2        bacon    12.0     pig
3     Pastrami     6.0     cow
4  corned beef     7.5     cow
5        Bacon     8.0     pig
6     pastrami     3.0     cow
7    honey ham     5.0     pig
8     nova lox     6.0  salmon
```

Replacing Values

- data = pd.Series([1., -999., 2., -999., -1000., 3.]); data
- data.replace(-999, np.nan)
- data.replace([-999, -1000], np.nan)
- data.replace([-999, -1000], [np.nan, 0]); data.replace({-999: np.nan, -1000: 0})

```
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

```
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

```
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
```

```
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

Renaming Axis Indexes

- data = pd.DataFrame(np.arange(12).reshape((3, 4)), index=['Ohio', 'Colorado', 'New York'], columns=['one', 'two', 'three', 'four']); data
- transform = lambda x: x[:4].upper()
- data.index.map(transform)
- Out[]: Index(['OHIO', 'COLO', 'NEW '], dtype='object')
- data.index = data.index.map(transform); data
- data.rename(index=str.title, columns=str.upper)
- data.rename(index={'OHIO': 'INDIANA'}, columns={'three': 'peekaboo'})

```
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
New York    8    9     10    11
```

```
      one  two  three  four
OHIO    0    1      2     3
COLO    4    5      6     7
NEW     8    9     10    11
```

```
      ONE  TWO  THREE  FOUR
Ohio    0    1      2     3
Colo    4    5      6     7
New     8    9     10    11
```

```
         one  two  peekaboo  four
INDIANA    0    1         2     3
COLO       4    5         6     7
NEW        8    9        10    11
```

Discretization and Binning

- ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
- bins = [18, 25, 35, 60, 100]
- cats = pd.cut(ages, bins); cats
- cats.codes
- Out[]: array([0, 0, 0, ..., 2, 2, 1], dtype=int8)
- cats.categories
- pd.value_counts(cats)

```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100],
(35, 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

```
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]]
              closed='right',
              dtype='interval[int64]')
```

```
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

Discretization and Binning

- pd.cut(ages, [18, 26, 36, 61, 100], right=False)
- group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']; pd.cut(ages, bins, labels=group_names)
- data = np.random.rand(20); pd.cut(data, 4, precision=2)

```
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100),
[36, 61), [36, 61), [26, 36)]
Length: 12
Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]

[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged,
MiddleAged, YoungAdult]
Length: 12
Categories (4, object): [MiddleAged < Senior < YoungAdult < Youth]

[(0.71, 0.93], (0.26, 0.49], (0.042, 0.26], (0.042, 0.26], (0.042, 0.26], ...,
(0.71, 0.93], (0.49, 0.71], (0.71, 0.93], (0.71, 0.93], (0.042, 0.26]]
Length: 20
Categories (4, interval[float64]): [(0.042, 0.26] < (0.26, 0.49] < (0.49, 0.71]
< (0.71, 0.93]]
```

Discretization and Binning

- data = np.random.randn(1000)
- cats = pd.qcut(data, 4); cats
- pd.value_counts(cats)
- pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])

```
(0.604, 3.237]      250
(-0.101, 0.604]     250
(-0.73, -0.101]     250
(-3.153, -0.73]     250
dtype: int64
```

```
[(-0.73, -0.101], (-0.73, -0.101], (0.604, 3.237], (-0.73, -0.101], (0.604,
3.237], ..., (0.604, 3.237], (0.604, 3.237], (0.604, 3.237], (0.604, 3.237],
(-3.153, -0.73]]
Length: 1000
Categories (4, interval[float64]): [(-3.153, -0.73] < (-0.73, -0.101] <
(-0.101, 0.604] < (0.604, 3.237]]
```

```
[(-1.332, -0.101], (-1.332, -0.101], (-0.101, 1.124], (-1.332, -0.101],
(-0.101, 1.124], ..., (1.124, 3.237], (-0.101, 1.124], (-0.101, 1.124],
(-0.101, 1.124], (-1.332, -0.101]]
Length: 1000
Categories (4, interval[float64]): [(-3.153, -1.332] < (-1.332, -0.101] <
(-0.101, 1.124] < (1.124, 3.237]]
```

Detecting and Filtering Outliers

- data = pd.DataFrame(np.random.randn(1000, 4)); data.describe()
- col = data[3]; col[np.abs(col) > 3]
- data[(np.abs(data) > 3).any(1)]
- data[np.abs(data) > 3] = np.sign(data) * 3; data.describe()

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.015703	0.004222	0.050356	0.007560
std	1.033428	1.010169	1.016482	1.031444
min	-3.311444	-2.783854	-3.162355	-3.303092
25%	-0.696384	-0.700096	-0.656921	-0.696223
50%	-0.008989	-0.002752	0.050823	-0.047989
75%	0.722099	0.688384	0.734656	0.736893
max	3.226275	2.982683	3.598494	2.873775

```
552   -3.285201
966   -3.303092
979   -3.127359
Name: 3, dtype: float64
```

	0	1	2	3
33	2.303525	-1.526105	-3.066275	0.626567
177	3.007859	0.559817	-1.632124	-0.815953
248	3.226275	0.117620	1.031936	0.121755
263	0.954599	-1.422743	3.598494	-0.597130
403	0.616683	1.558624	-3.162355	1.334664
521	2.975493	-0.912670	3.063208	0.216976
552	-0.670447	-0.589906	0.159256	-3.285201
576	0.652581	-1.246879	-3.092007	0.096121
690	0.966253	-0.130866	3.094115	-1.740745
773	-3.311444	-0.283092	-0.676722	0.634934
966	-0.179142	-0.478677	-1.709914	-3.303092
979	-0.516424	0.957380	-1.936210	-3.127359

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.015781	0.004222	0.049921	0.008275
std	1.031769	1.010169	1.013116	1.029261
min	-3.000000	-2.783854	-3.000000	-3.000000
25%	-0.696384	-0.700096	-0.656921	-0.696223
50%	-0.008989	-0.002752	0.050823	-0.047989
75%	0.722099	0.688384	0.734656	0.736893
max	3.000000	2.982683	3.000000	2.873775

Permutation and Random Sampling

- df = pd.DataFrame(np.arange(5 * 4).reshape(5, 4)); df
- sampler = np.random.permutation(5); sampler
- Out[]: array([4, 2, 1, 3, 0])
- df.take(sampler)
- choices = pd.Series ([5, 7, -1, 6, 4])
- draws = choices.sample (n=10, replace=True); draws

```
    0   1   2   3
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11
3  12  13  14  15
4  16  17  18  19

    0   1   2   3
4  16  17  18  19
2   8   9  10  11
1   4   5   6   7
3  12  13  14  15
0   0   1   2   3
```

```
0    5
3    6
1    7
3    6
2   -1
1    7
0    5
4    4
1    7
3    6
dtype: int64
```

Computing Indicator/Dummy Variables

- df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'], 'data1': range(6)}); df
- pd.get_dummies(df['key'])
- dummies = pd.get_dummies(df['key'], prefix='key'); dummies
- df_with_dummy = df[['data1']].join(dummies); df_with_dummy # index를 기준으로 left join (default)

	data1	key
0	0	b
1	1	b
2	2	a
3	3	c
4	4	a
5	5	b

	a	b	c
0	0	1	0
1	0	1	0
2	1	0	0
3	0	0	1
4	1	0	0
5	0	1	0

	key_a	key_b	key_c
0	0	1	0
1	0	1	0
2	1	0	0
3	0	0	1
4	1	0	0
5	0	1	0

	data1	key_a	key_b	key_c
0	0	0	1	0
1	1	0	1	0
2	2	1	0	0
3	3	0	0	1
4	4	1	0	0
5	5	0	1	0

Computing Indicator/Dummy Variables

- mnames = ['movie_id', 'title', 'genres']
- movies = pd.read_table('movies.dat', sep='::', header=None, names=mnames); movies
- movies.genres[0].split('|')
- Out[]: ['Animation', "Children's", 'Comedy']
- all_genres = []
- for x in movies.genres:
  - all_genres.extend(x.split('|'))
- genres = pd.unique(all_genres); genres

```
      movie_id                    title                        genres
0            1         Toy Story (1995)   Animation|Children's|Comedy
1            2           Jumanji (1995)  Adventure|Children's|Fantasy
2            3  Grumpier Old Men (1995)                Comedy|Romance
...        ...                      ...                           ...
3880      3950         Tigerland (2000)                         Drama
3881      3951  Two Family House (2000)                         Drama
3882      3952    Contender, The (2000)                Drama|Thriller

[3883 rows x 3 columns]
```

```
array(['Animation', "Children's", 'Comedy', ..., 'Mystery', 'Film-Noir'
       'Western'], dtype=object)
```

Computing Indicator/Dummy Variables

- zero_matrix = np.zeros((len(movies), len(genres)))
- dummies = pd.DataFrame(zero_matrix, columns=genres); dummies
- gen = movies.genres[0]
- dummies.columns.get_indexer(gen. split('|'))
- Out[]: array([0, 1, 2], dtype=int64)
- for i, gen in enumerate(movies.genres):
  - indices = dummies.columns.get_indexer(gen.split('|'))
  - dummies.iloc[i, indices] = 1
- movies_windic = movies.join(dummies.add_prefix('Genre_'))
- movies_windic.iloc[0]

```
      Animation  Children's  Comedy  Adventure  Fantasy  Romance  Drama  \
0           0.0         0.0     0.0        0.0      0.0      0.0    0.0
...         ...         ...     ...        ...      ...      ...    ...
3882        0.0         0.0     0.0        0.0      0.0      0.0    0.0

      Action  Crime  Thriller  Horror  Sci-Fi  Documentary  War  Musical  \
0        0.0    0.0       0.0     0.0     0.0          0.0  0.0      0.0
...      ...    ...       ...     ...     ...          ...  ...      ...
3882     0.0    0.0       0.0     0.0     0.0          0.0  0.0      0.0

      Mystery  Film-Noir  Western
0         0.0        0.0      0.0
...       ...        ...      ...
3882      0.0        0.0      0.0

[3883 rows x 18 columns]
```

```
movie_id                                        1
title                             Toy Story (1995)
genres               Animation|Children's|Comedy
Genre_Animation                                 1
Genre_Children's                                1
Genre_Comedy                                    1
Genre_Adventure                                 0
Genre_Fantasy                                   0
Genre_Romance                                   0
Genre_Drama                                     0
Genre_Action                                    0
Genre_Crime                                     0
Genre_Thriller                                  0
Genre_Horror                                    0
Genre_Sci-Fi                                    0
Genre_Documentary                               0
Genre_War                                       0
Genre_Musical                                   0
Genre_Mystery                                   0
Genre_Film-Noir                                 0
Genre_Western                                   0
Name: 0, dtype: object
```

Computing Indicator/Dummy Variables

- np.random.seed(12345)
- values = np.random.rand(10); values
- bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
- pd.get_dummies(pd.cut(values, bins))

	(0.0, 0.2]	(0.2, 0.4]	(0.4, 0.6]	(0.6, 0.8]	(0.8, 1.0]
0	0	0	0	0	1
1	0	1	0	0	0
2	1	0	0	0	0
3	0	1	0	0	0
4	0	0	1	0	0
5	0	0	1	0	0
6	0	0	0	0	1
7	0	0	0	1	0
8	0	0	0	1	0
9	0	0	0	1	0

```
array([ 0.92961609,  0.31637555,  0.18391881,  0.20456028,  0.56772503,
        0.5955447 ,  0.96451452,  0.6531771 ,  0.74890664,  0.65356987])
```

STRING MANIPULATION

String Object Methods

- val = 'a,b, guido'; val.split(',')
- Out[]: ['a', 'b', ' guido']
- pieces = [x.strip() for x in val.split(',')]; pieces
- Out[]: ['a', 'b', 'guido']
- first, second, third = pieces; first + '::' + second + '::' + third
- Out[]: 'a::b::guido'
- '::'.join(pieces)
- Out[]: 'a::b::guido'
- 'guido' in val
- Out[]: True
- val.index(',')
- Our[]:1
- val.find(':')
- Out:[]: -1
- val.index(':') #ValueError: substring not found
- val.count(',')
- Out[]: 2
- val.replace(',', '::')
- Out[]: 'a::b:: guido'

Regular expressions

- Regular expressions provide a flexible way to search or match string patterns in text
- The **re** module functions fall into three categories: pattern matching, substitution, and splitting
- text = "foo bar\t baz \tqux"
- re.split('\s+', text) # The **regex** describing one or more whitespace characters is \s+
- Out[]: ['foo', 'bar', 'baz', 'qux']

Regular expressions

- 정규 표현식에서 사용하는 메타 문자(meta characters)들
  - 메타 문자란? 원래 그 문자가 가진 뜻이 아닌 특별한 용도로 사용되는 문자
  - 정규 표현식에 메타 문자들이 사용되면 특별한 의미를 갖게 됨
  - . ^ $ * + ? { } [ ] \ | ( )
- 문자 클래스 [ ]
  - 문자 클래스로 만들어진 정규식은 "[와] 사이의 문자들과 매치"라는 의미
  - 문자 클래스 메타 문자도 사용할 수 있음
  - 예
    - [a-zA-Z] : 알파벳 모두
    - [0-9] : 숫자
- Dot(.): 줄바꿈 문자인 \n 를 제외한 모든 문자와 매치
- 반복: 메타문자 바로 앞에 있는 문자가
  - *: 0회 이상; +: 1회 이상; {m,n}: m회이상 n회 미만; ?: 0회 이상 1회 미만 반복 가능

메타 문자는 줄바꿈 문자인 \n을 제외한 모든 문자와 매치됨을 의미

Regular expressions

- escape sequence: 문자 앞에 \ 기호를 붙여 특수한 의미로 사용
  - \n: 줄바꿈
    - print('A man,\nA plan')
    - A man,
    - A plan
  - \t: 공백
    - print('\tabc')
    - 	abc
    - print('a\tbc')
    - a	bc

Regular expressions

- When you call re.split('\s+', text), the regular expression is first compiled, then its split method is called on the passed text
- regex = re.compile('\s+')
- regex.split(text)
- Out[]: ['foo', 'bar', 'baz', 'qux']
- regex.findall(text)
- Out[]: [' ', '\t ', ' \t']

Regular expressions

- **match** and **search** are closely related to **findall**
- While **findall** returns all matches in a string, **search** returns only the first match. More rigidly, **match** only matches at the beginning of the string
- text = """Dave dave@google.com
  - Steve steve@gmail.com
  - Rob rob@gmail.com
  - Ryan ryan@yahoo.com
  - """“
- pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'
- regex = re.compile(pattern, flags=re.IGNORECASE)
- regex.findall(text)
- Out[]: ['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']
- print(regex.match(text))
  - None
- regex.search(text)
- Out[20]: <_sre.SRE_Match object; span=(5, 20), match='dave@google.com'

Regular expressions

- pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'
- regex = re.compile(pattern, flags=re.IGNORECASE)
- regex.findall(text)

```
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

- print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))

```
Dave Username: dave, Domain: google, Suffix: com
Stev Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

Vectorized string functions in pandas

- data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com', 'Rob': 'rob@gmail.com', 'Wes': np.nan}
- data = pd.Series(data); data
- data.str.contains('gmail')
- data.str.findall(pattern, flags=re.IGNORECASE)
- matches = data.str.match(pattern, flags=re.IGNORECASE); matches

```
Dave     dave@google.com
Rob        rob@gmail.com
Steve    steve@gmail.com
Wes                  NaN
dtype: object
```

```
Dave     False
Rob       True
Steve     True
Wes        NaN
dtype: object
```

```
Dave     [(dave, google, com)]
Rob        [(rob, gmail, com)]
Steve    [(steve, gmail, com)]
Wes                        NaN
dtype: object
```

```
Dave     True
Rob      True
Steve    True
Wes       NaN
dtype: object
```

Vectorized string functions in pandas

- matches.str.get(1) # 파이썬 2.xxx 결과와 다름, 오류 같음
- matches.str[0] # 파이썬 2.xxx 결과와 다름, 오류 같음
- data.str[:5]

```
Dave     NaN
Rob      NaN
Steve    NaN
Wes      NaN
dtype: float64
```

```
Dave     NaN
Rob      NaN
Steve    NaN
Wes      NaN
dtype: float64
```

```
Dave      dave@
Rob       rob@g
Steve     steve
Wes         NaN
dtype: object
```

Vectorized string functions in pandas

- Python 2.xx의 결과와 비교

```
In [252]: matches
Out[252]:
Dave     ('dave', 'google', 'com')
Rob        ('rob', 'gmail', 'com')
Steve    ('steve', 'gmail', 'com')
Wes                            NaN
```

```
In [253]: matches.str.get(1)
Out[253]:
Dave     google
Rob       gmail
Steve     gmail
Wes         NaN
```

```
In [254]: matches.str[0]
Out[254]:
Dave      dave
Rob        rob
Steve    steve
Wes        NaN
```

Week 12:
Data Wrangling: Join, Combine, and Reshape

Hierarchical Indexing

Combining and Merging Data Sets

Reshaping and Pivoting

OUTLINE

HIERARCHICAL INDEXING

Hierarchical Indexing

- Hierarchical indexing is an important feature of pandas enabling you to have multiple (two or more) index levels on an axis
- data = Series(np.random.randn(10), index=[['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'd', 'd'], [1, 2, 3, 1, 2, 3, 1, 2, 2, 3]]); data

- data.index

```
a  1    1.642813
   2    0.919957
   3   -0.522089
b  1   -1.339057
   2    0.310053
   3    0.257015
c  1   -0.474308
   2    0.353321
d  2    0.780085
   3   -0.743241
dtype: float64
```

```
MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 1, 2, 0, 1, 1, 2]])
```

Hierarchical Indexing

- With a hierarchically-indexed object, partial indexing is possible, enabling you to concisely select subsets of the data
- data['b']; data['b':'c']; data.loc[['b', 'd']]; data.loc[:, 2]

```
1   -1.339057
2    0.310053
3    0.257015
dtype: float64
```

```
b  1   -1.339057
   2    0.310053
   3    0.257015
c  1   -0.474308
   2    0.353321
dtype: float64
```

```
b  1   -1.339057
   2    0.310053
   3    0.257015
d  2    0.780085
   3   -0.743241
dtype: float64
```

```
a    0.919957
b    0.310053
c    0.353321
d    0.780085
dtype: float64
```

Hierarchical Indexing

- Hierarchical indexing plays a critical role in reshaping data and group-based operations like forming a pivot table
- This data could be rearranged into a DataFrame using its **unstack** method
- data.unstack(); data.unstack().stack()

```
          1         2         3
a  1.642813  0.919957 -0.522089
b -1.339057  0.310053  0.257015
c -0.474308  0.353321       NaN
d       NaN  0.780085 -0.743241
```

```
a  1    1.642813
   2    0.919957
   3   -0.522089
b  1   -1.339057
   2    0.310053
   3    0.257015
c  1   -0.474308
   2    0.353321
d  2    0.780085
   3   -0.743241
dtype: float64
```

Hierarchical Indexing

- With a DataFrame, either axis can have a hierarchical index
- frame = DataFrame(np.arange(12).reshape((4, 3)), index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]], columns=[['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']]); frame
- frame.index.names = ['key1', 'key2']
- frame.columns.names = ['state', 'color']; frame
- frame['Ohio'] # Partial column indexing

```
     Ohio     Colorado
    Green Red    Green
a 1     0   1        2
  2     3   4        5
b 1     6   7        8
  2     9  10       11
```

```
state      Ohio     Colorado
color     Green Red    Green
key1 key2
a    1        0   1        2
     2        3   4        5
b    1        6   7        8
     2        9  10       11
```

```
color      Green  Red
key1 key2
a    1         0    1
     2         3    4
b    1         6    7
     2         9   10
```

Reordering and Sorting Levels

- The **swaplevel** takes two level numbers or names and returns a new object with the levels interchanged
- frame.swaplevel('key1', 'key2')

state		Ohio		Colorado
color		Green	Red	Green
key2	key1			
1	a	0	1	2
2	a	3	4	5
1	b	6	7	8
2	b	9	10	11

- **sortlevel** sorts the data using only the values in a single level.
- frame.sort_index (axis=1); frame.swaplevel(0, 1).sort_index (axis=0)

state		Colorado	Ohio	
color		Green	Green	Red
key1	key2			
a	1	2	0	1
	2	5	3	4
b	1	8	6	7
	2	11	9	10

state		Ohio		Colorado
color		Green	Red	Green
key2	key1			
1	a	0	1	2
	b	6	7	8
2	a	3	4	5
	b	9	10	11

Summary Statistics by Level

- Many descriptive and summary statistics on DataFrame and Series have a **level** option in which you can specify the level you want to aggregate by on a particular axis
- frame.sum(level='key2'); frame.sum(level='color', axis=1)

```
state  Ohio     Colorado
color Green Red    Green
key2
1          6   8       10
2         12  14       16
```

```
color       Green  Red
key1 key2
a    1          2    1
     2          8    4
b    1         14    7
     2         20   10
```

Using a DataFrame's Columns

- DataFrame's **set_index** function will create a new DataFrame using one or more of its columns as the index
- **reset_index** does the opposite of **set_index**; the hierarchical index levels are moved into the columns

- frame = DataFrame({'a': range(7), 'b': range(7, 0, -1), 'c': ['one', 'one', 'one', 'two', 'two', 'two', 'two'], 'd': [0, 1, 2, 0, 1, 2, 3]}); frame

	a	b	c	d
0	0	7	one	0
1	1	6	one	1
2	2	5	one	2
3	3	4	two	0
4	4	3	two	1
5	5	2	two	2
6	6	1	two	3

Using a DataFrame's Columns

- frame2=frame.set_index(['c', 'd']); frame2
- frame.set_index(['c', 'd'], drop=False)
- frame2.reset_index()

```
         a  b
c   d
one 0    0  7
    1    1  6
    2    2  5
two 0    3  4
    1    4  3
    2    5  2
    3    6  1
```

```
         a  b    c    d
c   d
one 0    0  7    one  0
    1    1  6    one  1
    2    2  5    one  2
two 0    3  4    two  0
    1    4  3    two  1
    2    5  2    two  2
    3    6  1    two  3
```

```
     c    d  a  b
0  one    0  0  7
1  one    1  1  6
2  one    2  2  5
3  two    0  3  4
4  two    1  4  3
5  two    2  5  2
6  two    3  6  1
```

COMBINING AND MERGING DATA SETS

Database-style DataFrame Merges

- Data contained in pandas objects can be combined together in a number of built-in ways
  - pandas.**merge** connects rows in DataFrames based on one or more keys
  - pandas.**concat** glues or stacks together objects along an axis
  - **combine_first** instance method enables splicing together overlapping data to fill in missing values in one object with values from another

Database-style DataFrame Merges

- Merge or join operations combine data sets by linking rows using one or more keys
- df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], 'data1': range(7)}); df1
- df2 = pd.DataFrame({'key': ['a', 'b', 'd'], 'data2': range(3)}); df2

```
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   a
6      6   b
```

```
   data2 key
0      0   a
1      1   b
2      2   d
```

Database-style DataFrame Merges

- This is an example of a many-to-one merge situation
- pd.merge(df1, df2) ; pd.merge(df1, df2, on='key')

```
   dat a1 key  data2
0        0   b      1
1        1   b      1
2        6   b      1
3        2   a      0
4        4   a      0
5        5   a      0
```

```
   data1 lkey  data2 rkey
0      0    b      1    b
1      1    b      1    b
2      6    b      1    b
3      2    a      0    a
4      4    a      0    a
5      5    a      0    a
```

- If the column names are different in each object, you can specify them separately
- df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], 'data1': range(7)})
- df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'], 'data2': range(3)})
- pd.merge(df3, df4, left_on='lkey', right_on='rkey')

Database-style DataFrame Merges

- By default merge does an 'inner' join; the keys in the result are the intersection
- Other possible options are 'left' , 'right' , and 'outer' . The outer join takes the union of the keys, combining the effect of applying both left and right joins

- pd.merge(df1, df2, how='outer')

```
   dat_a1 key  data2
0     0.0   b    1.0
1     1.0   b    1.0
2     6.0   b    1.0
3     2.0   a    0.0
4     4.0   a    0.0
5     5.0   a    0.0
6     3.0   c    NaN
7     NaN   d    2.0
```

Database-style DataFrame Merges

- Many-to-many joins form the Cartesian product of the rows. The join method only affects the distinct key values appearing in the result
- df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'], 'data1': range(6)}); df1
- df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'], 'data2': range(5)}); df2

	data1	key
0	0	b
1	1	b
2	2	a
3	3	c
4	4	a
5	5	b

	data2	key
0	0	a
1	1	b
2	2	a
3	3	b
4	4	d

- pd.merge(df1, df2, on='key', how='left')
- pd.merge(df1, df2, how='inner')

	data1	key	data2
0	0	b	1.0
1	0	b	3.0
2	1	b	1.0
3	1	b	3.0
4	2	a	0.0
5	2	a	2.0
6	3	c	NaN
7	4	a	0.0
8	4	a	2.0
9	5	b	1.0
10	5	b	3.0

	data1	key	data2
0	0	b	1
1	0	b	3
2	1	b	1
3	1	b	3
4	5	b	1
5	5	b	3
6	2	a	0
7	2	a	2
8	4	a	0
9	4	a	2

Database-style DataFrame Merges

- To merge with multiple keys, pass a list of column names
- left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'], 'key2': ['one', 'two', 'one'], 'lval': [1, 2, 3]}); left
- right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'], 'key2': ['one', 'one', 'one', 'two'], 'rval': [4, 5, 6, 7]}); right

```
  key1 key2  lval
0  foo  one     1
1  foo  two     2
2  bar  one     3
```

```
  key1 key2  rval
0  foo  one     4
1  foo  one     5
2  bar  one     6
3  bar  two     7
```

- pd.merge(left, right, on=['key1', 'key2'], how='outer')

```
   key1 key2  lval  rval
0   foo  one   1.0   4.0
1   foo  one   1.0   5.0
2   foo  two   2.0   NaN
3   bar  one   3.0   6.0
4   bar  two   NaN   7.0
```

Database-style DataFrame Merges

- A last issue to consider in merge operations is the treatment of overlapping column names. While you can address the overlap manually, merge has a **suffixes** option for specifying strings to append to overlapping names in the left and right DataFrame objects

- pd.merge(left, right, on='key1')
- pd.merge(left, right, on='key1', suffixes=('_left', '_right'))

```
  key1 key2_x  lval key2_y  rval
0  foo    one     1    one     4
1  foo    one     1    one     5
2  foo    two     2    one     4
3  foo    two     2    one     5
4  bar    one     3    one     6
5  bar    one     3    two     7
```

```
  key1 key2_left  lval key2_right  rval
0  foo       one     1        one     4
1  foo       one     1        one     5
2  foo       two     2        one     4
3  foo       two     2        one     5
4  bar       one     3        one     6
5  bar       one     3        two     7
```

join method

- left.join(right.set_index(['key1', 'key2']), on=['key1', 'key2'], how='outer') # 두 object가 key1과 key2를 공유할 때
- left.join(right.set_index('key1'), on='key1', how='inner', lsuffix='_left', rsuffix='_right')

	key1	key2	lval	rval
0	foo	one	1.0	4.0
0	foo	one	1.0	5.0
1	foo	two	2.0	NaN
2	bar	one	3.0	6.0
2	bar	two	NaN	7.0

	key1	key2_left	lval	key2_right	rval
0	foo	one	1	one	4
0	foo	one	1	one	5
1	foo	two	2	one	4
1	foo	two	2	one	5
2	bar	one	3	one	6
2	bar	one	3	two	7

Merging on Index

- left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'], 'value': range(6)}); left1
- right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b']); right1

```
   key  value
0    a      0
1    b      1
2    a      2
3    a      3
4    b      4
5    c      5
```

```
   group_val
a        3.5
b        7.0
```

Merging on Index

- In some cases, the merge key or keys in a DataFrame will be found in its index. In this case, you can pass **left_index=True** or **right_index=True** (or both)
- pd.merge(left1, right1, left_on='key', right_index=True)

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0

- You can form the union of them with an outer join
- pd.merge(left1, right1, left_on='key', right_index=True, how='outer')

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0
5	c	5	NaN

join method

- left1.join(right1, on='key', how='inner') # object right1이 index를 가질 때
- left1.join(right1, on='key', how='outer')

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0
5	c	5	NaN

Merging on Index

- lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'], 'key2': [2000, 2001, 2002, 2001, 2002], 'data': np.arange(5.)}); lefth
- righth = pd.DataFrame(np.arange(12).reshape((6, 2)), index=[['Nevada', 'Nevada', 'Ohio', 'Ohio', 'Ohio', 'Ohio'], [2001, 2000, 2000, 2000, 2001, 2002]], columns=['event1', 'event2']); righth

	data	key1	key2
0	0.0	Ohio	2000
1	1.0	Ohio	2001
2	2.0	Ohio	2002
3	3.0	Nevada	2001
4	4.0	Nevada	2002

		event1	event2
Nevada	2001	0	1
	2000	2	3
Ohio	2000	4	5
	2000	6	7
	2001	8	9
	2002	10	11

Merging on Index

- With hierarchically-indexed data, things are more complicated
- pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)
- pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True, how='outer')

	data	key1	key2	event1	event2
0	0.0	Ohio	2000	4	5
0	0.0	Ohio	2000	6	7
1	1.0	Ohio	2001	8	9
2	2.0	Ohio	2002	10	11
3	3.0	Nevada	2001	0	1

	data	key1	key2	event1	event2
0	0.0	Ohio	2000	4.0	5.0
0	0.0	Ohio	2000	6.0	7.0
1	1.0	Ohio	2001	8.0	9.0
2	2.0	Ohio	2002	10.0	11.0
3	3.0	Nevada	2001	0.0	1.0
4	4.0	Nevada	2002	NaN	NaN
4	NaN	Nevada	2000	2.0	3.0

join method: slide

- lefth.join(righth, on=['key1', 'key2'], how='inner') # object righth가 hierarchical index를 가질 때
- lefth.join(righth, on=['key1', 'key2'], how='outer')

	data	key1	key2	event1	event2
0	0.0	Ohio	2000	4	5
0	0.0	Ohio	2000	6	7
1	1.0	Ohio	2001	8	9
2	2.0	Ohio	2002	10	11
3	3.0	Nevada	2001	0	1

	data	key1	key2	event1	event2
0	0.0	Ohio	2000	4.0	5.0
0	0.0	Ohio	2000	6.0	7.0
1	1.0	Ohio	2001	8.0	9.0
2	2.0	Ohio	2002	10.0	11.0
3	3.0	Nevada	2001	0.0	1.0
4	4.0	Nevada	2002	NaN	NaN
4	NaN	Nevada	2000	2.0	3.0

Merging on Index

- Using the indexes of both sides of the merge is also possible
- left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]], index=['a', 'c', 'e'], columns=['Ohio', 'Nevada']); left2
- right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]], index=['b', 'c', 'd', 'e'], columns=['Missouri', 'Alabama']); right2

	Ohio	Nevada
a	1.0	2.0
c	3.0	4.0
e	5.0	6.0

	Missouri	Alabama
b	7.0	8.0
c	9.0	10.0
d	11.0	12.0
e	13.0	14.0

Merging on Index

- pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
- DataFrame has a more convenient **join** instance for merging by index. It can also be used to combine together many DataFrame objects having the same or similar indexes but non-overlapping columns
- left2.join(right2, how='outer')

	Ohio	Nevada	Missouri	Alabama
a	1.0	2.0	NaN	NaN
b	NaN	NaN	7.0	8.0
c	3.0	4.0	9.0	10.0
d	NaN	NaN	11.0	12.0
e	5.0	6.0	13.0	14.0

Merging on Index

- DataFrame's **join** method performs a left join on the join keys. It supports joining the index of the passed DataFrame on one of the columns of the calling DataFrame
- left1.join(right1, on='key')

	key	value	group_val
0	a	0	3.5
1	b	1	7.0
2	a	2	3.5
3	a	3	3.5
4	b	4	7.0
5	c	5	NaN

Merging on Index

- another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]], index=['a', 'c', 'e', 'f'], columns=['New York', 'Oregon'])
- For simple index-on-index merges, you can pass a list of DataFrames to join
- left2.join([right2, another])
- left2.join([right2, another], how='outer')

	New York	Oregon
a	7.0	8.0
c	9.0	10.0
e	11.0	12.0
f	16.0	17.0

	Ohio	Nevada	Missouri	Alabama	New York	Oregon
a	1.0	2.0	NaN	NaN	7.0	8.0
c	3.0	4.0	9.0	10.0	9.0	10.0
e	5.0	6.0	13.0	14.0	11.0	12.0

	Ohio	Nevada	Missouri	Alabama	New York	Oregon
a	1.0	2.0	NaN	NaN	7.0	8.0
b	NaN	NaN	7.0	8.0	NaN	NaN
c	3.0	4.0	9.0	10.0	9.0	10.0
d	NaN	NaN	11.0	12.0	NaN	NaN
e	5.0	6.0	13.0	14.0	11.0	12.0
f	NaN	NaN	NaN	NaN	16.0	17.0

Concatenating Along an Axis

- Suppose we have three Series with no index overlap
- s1 = pd.Series([0, 1], index=['a', 'b']); s1
- s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e']); s2
- s3 = pd.Series([5, 6], index=['f', 'g']); s3
- Calling **concat** with these object in a list glues together the values and indexes
- pd.concat([s1, s2, s3])

```
a    0
b    1
dtype: int64
```

```
c    2
d    3
e    4
dtype: int64
```

```
f    5
g    6
dtype: int64
```

```
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: int64
```

Concatenating Along an Axis

- If you pass axis=1, the result will be a DataFrame (axis=1 is the columns)
- pd.concat([s1, s2, s3], axis=1)
- s4 = pd.concat([s1, s3]); s4
- You can intersect them by passing join='inner'
- pd.concat([s1, s4], axis=1, join='inner')
- pd.concat([s1, s4], axis=1, join='outer')

```
     0    1    2
a  0.0  NaN  NaN
b  1.0  NaN  NaN
c  NaN  2.0  NaN
d  NaN  3.0  NaN
e  NaN  4.0  NaN
f  NaN  NaN  5.0
g  NaN  NaN  6.0
```

```
a    0
b    1
f    5
g    6
dtype: int64
```

```
   0  1
a  0  0
b  1  1
```

```
     0  1
a  0.0  0
b  1.0  1
f  NaN  5
g  NaN  6
```

Concatenating Along an Axis

- You can even specify the axes to be used on the other axes with **join_axes**
- pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
- Suppose you wanted to create a hierarchical index on the concatenation axis
- result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three']); result
- result.unstack()

```
     0    1
a  0.0  0.0
c  NaN  NaN
b  1.0  1.0
e  NaN  NaN
```

```
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6
dtype: int64
```

```
          a    b    f    g
one     0.0  1.0  NaN  NaN
two     0.0  1.0  NaN  NaN
three   NaN  NaN  5.0  6.0
```

Concatenating Along an Axis

- In the case of combining Series along axis=1, the keys become the DataFrame column headers
- pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
- df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'], columns=['one', 'two']); df1
- df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'], columns=['three', 'four']); df2
- pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])

```
   one  two  three
a  0.0  NaN    NaN
b  1.0  NaN    NaN
c  NaN  2.0    NaN
d  NaN  3.0    NaN
e  NaN  4.0    NaN
f  NaN  NaN    5.0
g  NaN  NaN    6.0
```

```
   one  two
a    0    1
b    2    3
c    4    5
```

```
   three  four
a      5     6
c      7     8
```

```
     level1     level2
        one two  three four
a         0   1    5.0  6.0
b         2   3    NaN  NaN
c         4   5    7.0  8.0
```

Concatenating Along an Axis

- If you pass a dict of objects instead of a list, the dict's keys will be used for the **keys** option
- pd.concat({'level1': df1, 'level2': df2}, axis=1)
- We can name the created axis levels with **names** argument
- pd.concat([df1, df2], axis=1, keys=['level1', 'level2'], names=['upper', 'lower'])

```
  level1     level2
     one two  three four
a      0   1    5.0  6.0
b      2   3    NaN  NaN
c      4   5    7.0  8.0

upper level1     level2
lower    one two  three four
a          0   1    5.0  6.0
b          2   3    NaN  NaN
c          4   5    7.0  8.0
```

Concatenating Along an Axis

- A last consideration concerns DataFrames in which the row index is not meaningful in the context of the analysis
- df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd']); df1
- df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a']); df2
- pd.concat([df1, df2], ignore_index=True)

```
          a         b         c         d
0 -0.451682 -0.189586 -0.407904 -1.575067
1 -1.343365  0.060235  2.041810  1.704309
2  0.558765 -1.778323 -0.015996 -0.386477
          b         d         a
0 -1.344977  0.975734 -0.029633
1  0.170530 -1.631717 -0.382574
          a         b         c         d
0 -0.451682 -0.189586 -0.407904 -1.575067
1 -1.343365  0.060235  2.041810  1.704309
2  0.558765 -1.778323 -0.015996 -0.386477
3 -0.029633 -1.344977       NaN  0.975734
4 -0.382574  0.170530       NaN -1.631717
```

concat method

- pd.concat([df1, df2]) # index를 고려한 경우
- pd.concat([df1, df2], axis=1) # 열로 병합

```
          a         b         c         d
0 -0.249920 -1.368020  0.366403 -0.892764
1 -0.551183  0.692886  0.170155  1.815337
2 -1.088054  0.864913 -0.654142  1.127031
0  0.378386  1.017346       NaN  1.288937
1 -0.311037  0.767638       NaN  0.748961
```

```
          a         b         c         d         b         d         a
0 -0.249920 -1.368020  0.366403 -0.892764  1.017346  1.288937  0.378386
1 -0.551183  0.692886  0.170155  1.815337  0.767638  0.748961 -0.311037
2 -1.088054  0.864913 -0.654142  1.127031       NaN       NaN       NaN
```

Combining Data with Overlap

- a = pd.Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan], index=['f', 'e', 'd', 'c', 'b', 'a']); a
- b = pd.Series(np.arange(len(a), dtype=np.float64), index=['f', 'e', 'd', 'c', 'b', 'a']); b
- b[-1] = np.nan; b

```
f    NaN        f    0.0        f    0.0
e    2.5        e    1.0        e    1.0
d    NaN        d    2.0        d    2.0
c    3.5        c    3.0        c    3.0
b    4.5        b    4.0        b    4.0
a    NaN        a    5.0        a    NaN
dtype: float64  dtype: float64  dtype: float64
```

- np.where(pd.isnull(a), b, a)
- Out[]: array([0. , 2.5, 2. , 3.5, 4.5, nan])
- Series has a **combine_first** method, which performs the equivalent of this operation plus data alignment
- b[:-2].combine_first(a[2:])

```
a    NaN
b    4.5
c    3.0
d    2.0
e    1.0
f    0.0
dtype: float64
```

combine_first method

- b[:-2]; a[2:]
- b[:-2].combine_first(a[2:]) # left에 있는 object의 값이 null이면 right object의 값으로 대체, 그렇지 않으면 left object 값을 그대로 유지
- a[2:].combine_first(b[:-2]

```
f    0.0
e    1.0
d    2.0
c    3.0
dtype: float64
```

```
d    NaN
c    3.5
b    4.5
a    NaN
dtype: float64
```

```
a    NaN
b    4.5
c    3.0
d    2.0
e    1.0
f    0.0
dtype: float64
```

```
a    NaN
b    4.5
c    3.5
d    2.0
e    1.0
f    0.0
dtype: float64
```

Combining Data with Overlap

- df1 = pd.DataFrame({'a': [1., np.nan, 5., np.nan], 'b': [np.nan, 2., np.nan, 6.], 'c': range(2, 18, 4)}); df1
- df2 = pd.DataFrame({'a': [5., 4., np.nan, 3., 7.], 'b': [np.nan, 3., 4., 6., 8.]}); df2
- With DataFrames, **combine_first** naturally does the same thing column by column, so you can think of it as "patching" missing data in the calling object with data from the object you pass
- df1.combine_first(df2)

	a	b	c
0	1.0	NaN	2
1	NaN	2.0	6
2	5.0	NaN	10
3	NaN	6.0	14

	a	b
0	5.0	NaN
1	4.0	3.0
2	NaN	4.0
3	3.0	6.0
4	7.0	8.0

	a	b	c
0	1.0	NaN	2.0
1	4.0	2.0	6.0
2	5.0	4.0	10.0
3	3.0	6.0	14.0
4	7.0	8.0	NaN

combine_first method

- df2.combine_first(df1)

```
     a    b     c
0  5.0  NaN   2.0
1  4.0  3.0   6.0
2  5.0  4.0  10.0
3  3.0  6.0  14.0
4  7.0  8.0   NaN
```

RESHAPING AND PIVOTING

Reshaping with Hierarchical Indexing

- Hierarchical indexing provides a consistent way to rearrange data in a DataFrame
  - stack : this "rotates" or pivots from the columns in the data to the rows
  - unstack : this pivots from the rows into the columns
- data = pd.DataFrame(np.arange(6).reshape((2, 3)), index=pd.Index(['Ohio', 'Colorado'], name='state'), columns=pd.Index(['one', 'two', 'three'], name='number')); data

```
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

- s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])
- s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])
- data2 = pd.concat([s1, s2], keys=['one', 'two']); data2

```
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: int64
```

Reshaping with Hierarchical Indexing

- result = data.stack(); result
- result.unstack()
- result.unstack(0); result.unstack('state')
- data2.unstack()
- data2.unstack(0)
- data2.unstack().stack()
- data2.unstack().stack(dropna=False)

```
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
dtype: int32
```

```
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

```
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5
```

```
       a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

```
   one  two
a  0.0  NaN
b  1.0  NaN
c  2.0  4.0
d  3.0  5.0
e  NaN  6.0
```

```
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
     e    NaN
two  a    NaN
     b    NaN
     c    4.0
     d    5.0
     e    6.0
dtype: float64
```

```
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
     e    NaN
two  a    NaN
     b    NaN
     c    4.0
     d    5.0
     e    6.0
dtype: float64
```

Reshaping with Hierarchical Indexing

- When unstacking in a DataFrame, the level unstacked becomes the lowest level in the result
- df = pd.DataFrame({'left': result, 'right': result + 5}, columns=pd.Index(['left', 'right'], name='side'));df
- df.unstack()
- df.unstack('state')

```
side                left  right
state    number
Ohio     one           0      5
         two           1      6
         three         2      7
Colorado one           3      8
         two           4      9
         three         5     10
```

```
side     left            right
number    one two three    one two three
state
Ohio        0   1     2      5   6     7
Colorado    3   4     5      8   9    10
```

```
side   left           right
state  Ohio Colorado   Ohio Colorado
number
one       0        3     5        8
two       1        4     6        9
three     2        5     7       10
```

Pivoting “long” to “wide” Format

- A common way to store multiple time series in databases and CSV is in so-called long or stacked format
- data = pd.read_csv("macrodata.csv"); data
- periods = pd.PeriodIndex(year=data.year, quarter=data.quarter, name='date')
- columns = pd.Index(['realgdp', 'infl', 'unemp'], name='item')
- data = data.reindex(columns=columns); data
- data.index = periods.to_timestamp('D', 'end'); data

```
      year  quarter    realgdp  realcons   realinv  realgovt  realdpi  \
0   1959.0      1.0   2710.349    1707.4   286.898   470.045   1886.9
1   1959.0      2.0   2778.801    1733.7   310.859   481.301   1919.7
2   1959.0      3.0   2775.488    1751.8   289.226   491.260   1916.4
..     ...      ...        ...       ...       ...       ...      ...
200 2009.0      1.0  12925.410    9209.2  1558.494   996.287   9926.4
201 2009.0      2.0  12901.504    9189.0  1456.678  1023.528  10077.5
202 2009.0      3.0  12990.341    9256.0  1486.398  1044.088  10040.6

         cpi      m1  tbilrate  unemp      pop  infl  realint
0     28.980   139.7      2.82    5.8  177.146  0.00     0.00
1     29.150   141.7      3.08    5.1  177.830  2.34     0.74
2     29.350   140.5      3.82    5.3  178.657  2.74     1.09
..       ...     ...       ...    ...      ...   ...      ...
200  212.671  1592.8      0.22    8.1  306.547  0.94    -0.71
201  214.469  1653.6      0.18    9.2  307.226  3.37    -3.19
202  216.385  1673.9      0.12    9.6  308.013  3.56    -3.44
```

```
item     realgdp  infl  unemp
0       2710.349  0.00    5.8
1       2778.801  2.34    5.1
2       2775.488  2.74    5.3
..           ...   ...    ...
200   12925.410  0.94    8.1
201   12901.504  3.37    9.2
202   12990.341  3.56    9.6
```

```
item                realgdp  infl  unemp
date
1959-03-31        2710.349  0.00    5.8
1959-06-30        2778.801  2.34    5.1
1959-09-30        2775.488  2.74    5.3
...                    ...   ...    ...
2009-03-31       12925.410  0.94    8.1
2009-06-30       12901.504  3.37    9.2
2009-09-30       12990.341  3.56    9.6
```

Pivoting “long” to “wide” Format

- data.stack()
- data.stack().reset_index()
- data.stack().reset_index().rename(columns={0:'value'})
- ldata = data.stack().reset_index().rename(columns={0:'value'})

```
date        item
1959-03-31  realgdp     2710.349
            infl           0.000
            unemp          5.800
                            ...
2009-09-30  realgdp    12990.341
            infl           3.560
            unemp          9.600
Length: 609, dtype: float64
```

```
                         date     item          0
0   1959-03-31  realgdp   2710.349
1   1959-03-31     infl      0.000
2   1959-03-31    unemp      5.800
..         ...      ...        ...
606 2009-09-30  realgdp  12990.341
607 2009-09-30     infl      3.560
608 2009-09-30    unemp      9.600

[609 rows x 3 columns]
```

```
           date     item      value
0    1959-03-31  realgdp   2710.349
1    1959-03-31     infl      0.000
2    1959-03-31    unemp      5.800
..          ...      ...        ...
606  2009-09-30  realgdp  12990.341
607  2009-09-30     infl      3.560
608  2009-09-30    unemp      9.600
```

Pivoting “long” to “wide” Format

- pivoted = ldata.pivot('date', 'item', 'value'); pivoted
- ldata['value2'] = np.random.randn(len(ldata)); ldata
- pivoted2 = ldata.pivot('date', 'item'); pivoted2

```
item        infl    realgdp  unemp
date
1959-03-31  0.00   2710.349    5.8
1959-06-30  2.34   2778.801    5.1
1959-09-30  2.74   2775.488    5.3
...          ...         ...    ...
2009-03-31  0.94  12925.410    8.1
2009-06-30  3.37  12901.504    9.2
2009-09-30  3.56  12990.341    9.6

[203 rows x 3 columns]
```

```
          date     item      value    value2
0   1959-03-31  realgdp   2710.349  0.846966
1   1959-03-31     infl      0.000 -0.780523
2   1959-03-31    unemp      5.800 -1.137184
..         ...      ...        ...       ...
606 2009-09-30  realgdp  12990.341 -1.628197
607 2009-09-30     infl      3.560 -0.818834
608 2009-09-30    unemp      9.600  0.820156

[609 rows x 4 columns]
```

```
           value                       value2
item        infl    realgdp unemp      infl    realgdp     unemp
date
1959-03-31  0.00   2710.349   5.8 -0.780523  0.846966 -1.137184
1959-06-30  2.34   2778.801   5.1  0.110029  1.440342  1.096601
1959-09-30  2.74   2775.488   5.3 -0.330139  0.065843  0.217372
...          ...        ...   ...       ...       ...       ...
2009-03-31  0.94  12925.410   8.1 -0.235259 -0.958347  0.942189
2009-06-30  3.37  12901.504   9.2  0.202360  0.412192 -2.574395
2009-09-30  3.56  12990.341   9.6 -0.818834 -1.628197  0.820156

[203 rows x 6 columns]
```

Pivoting "wide" to "long" Format

- A inversion operation to pivot for DataFrames is pandas.**melt**
- df = pd.DataFrame({'key': ['foo', 'bar', 'baz'], 'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})
- melted = pd.melt(df, ['key']); melted # use 'key' as the group indicator
- reshaped = melted.pivot('key', 'variable', 'value')
- reshaped.reset_index()
- pd.melt(df, id_vars=['key'], value_vars=['A', 'B'])

```
   A  B  C  key
0  1  4  7  foo
1  2  5  8  bar
2  3  6  9  baz
```

```
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
..  ...      ...    ...
6  foo        C      7
7  bar        C      8
8  baz        C      9

[9 rows x 3 columns]
```

```
variable  key  A  B  C
0         bar  2  5  8
1         baz  3  6  9
2         foo  1  4  7
```

```
variable    A  B  C
key
bar         2  5  8
baz         3  6  9
foo         1  4  7
```

```
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
```

Week 13:
Plotting and Visualization

A Brief matplotlib API Primer

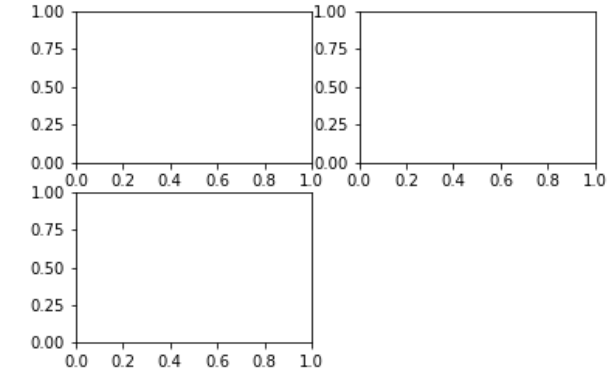
Plotting with pandas and seaborn

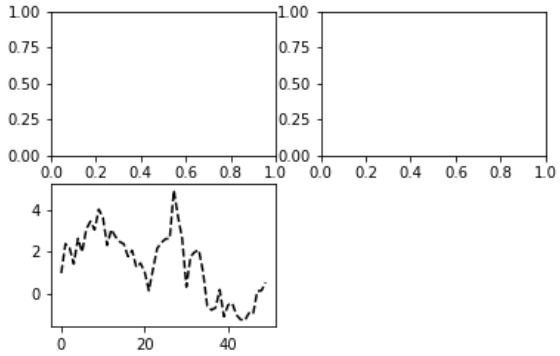
OUTLINE

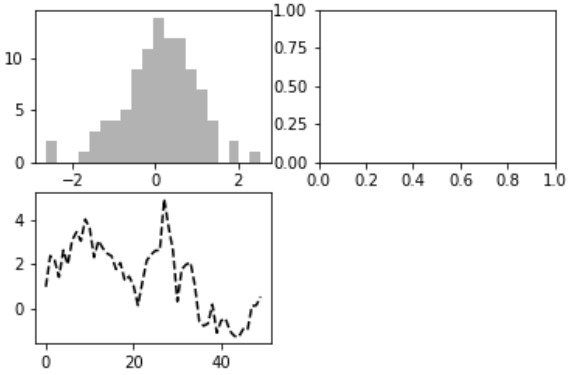
A BRIEF MATPLOTLIB API PRIMER

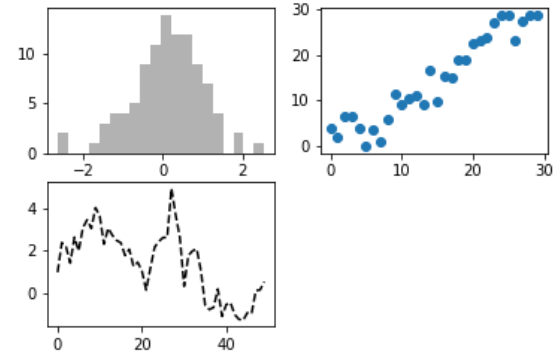
Figures and Subplots

- import matplotlib.pyplot as plt
- %matplotlib
- fig = plt.figure()
- ax1 = fig.add_subplot(2, 2, 1)
- ax2 = fig.add_subplot(2, 2, 2)
- ax3 = fig.add_subplot(2, 2, 3)
- plt.plot(np.random.randn(50).cumsum(), 'k--')
- ax1.hist(np.random.randn(100), bins=20, color='k', alpha=0.3)
- ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.randn(30))



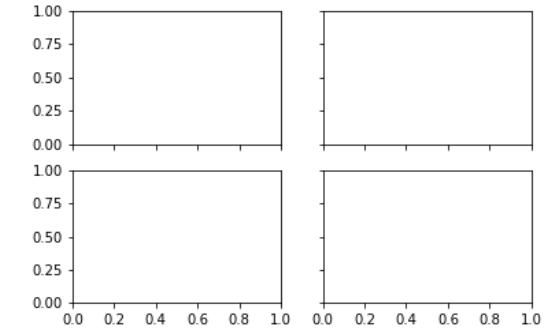


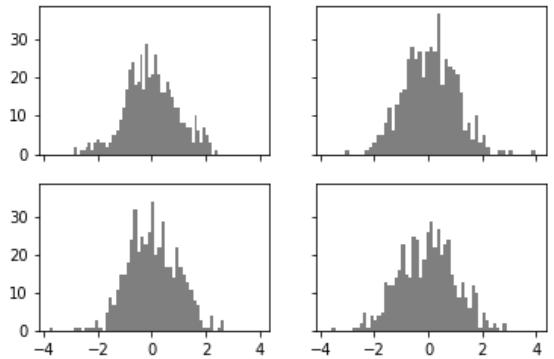


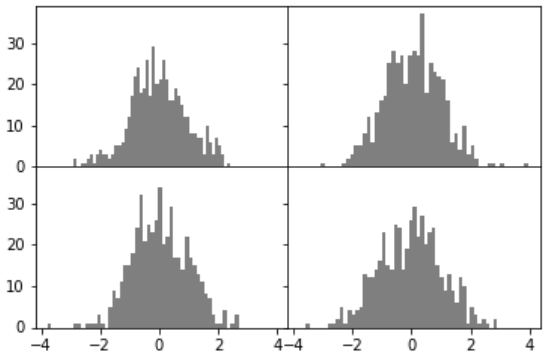


Figures and Subplots

- fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
- for i in range(2):
  - for j in range(2):
    - axes[i, j].hist(np.random.randn(500), bins=50, color='k', alpha=0.5)
- plt.subplots_adjust(wspace=0, hspace=0)

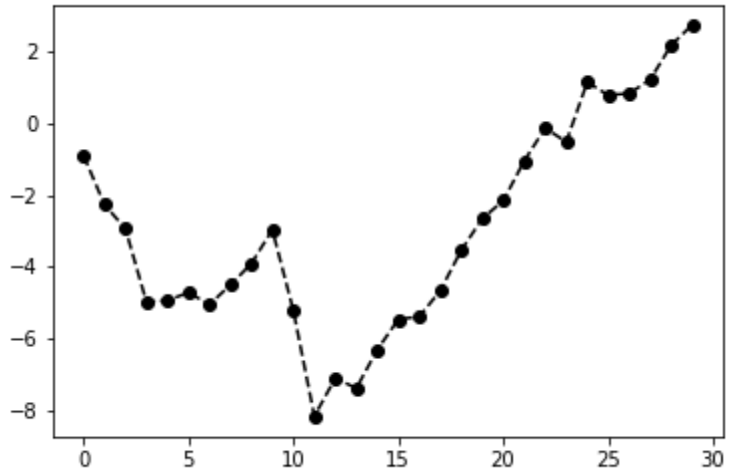


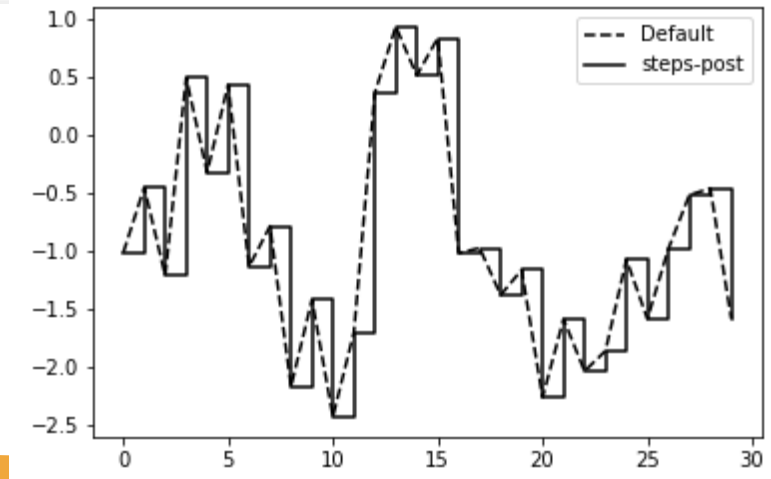




Colors, Markers, and Line Styles

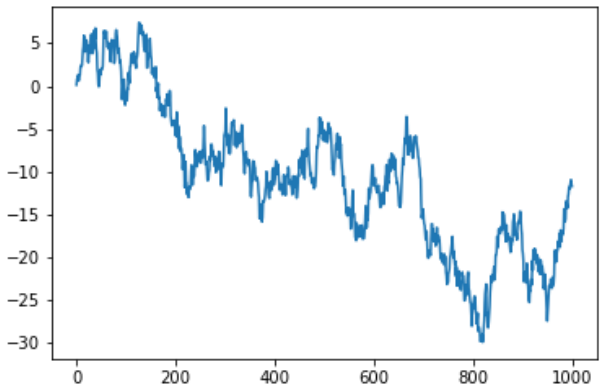
- plt.figure()
- plt.plot(np.random.randn(30).cumsum(), 'ko--') # equivalent to plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')
- data = np.random.randn(30).cumsum()
- plt.plot(data, 'k--', label='Default')
- plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')
- plt.legend(loc='best')

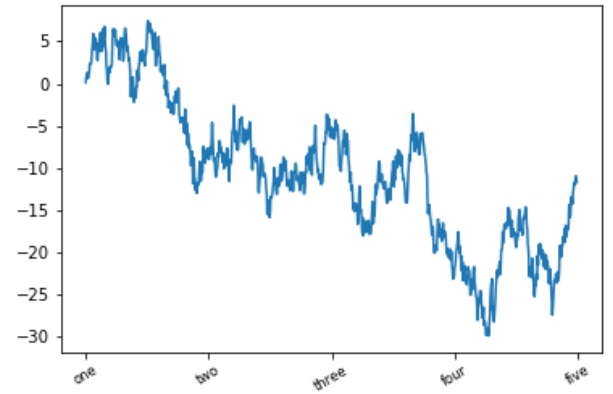


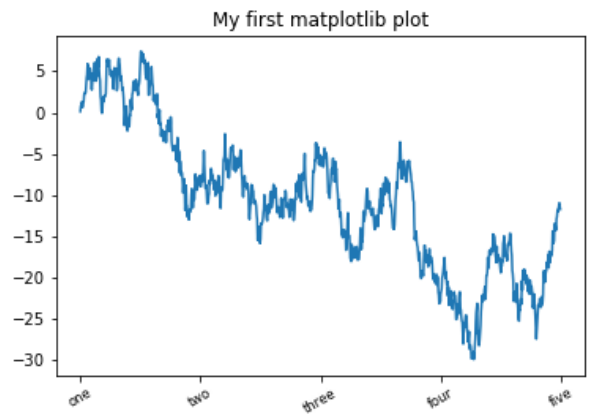


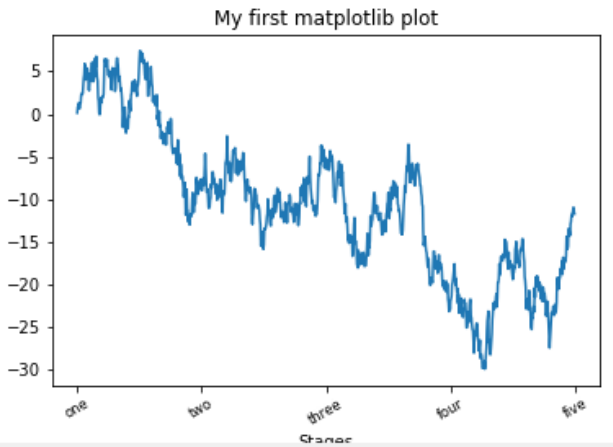
Ticks, Labels, and Legends

- fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)
- ax.plot(np.random.randn(1000).cumsum())
- ax.set_xticks([0, 250, 500, 750, 1000]); ax.set_xticklabels(['one', 'two', 'three', 'four', 'five'], rotation=30, fontsize='small')
- ax.set_title('My first matplotlib plot')
- ax.set_xlabel('Stages')



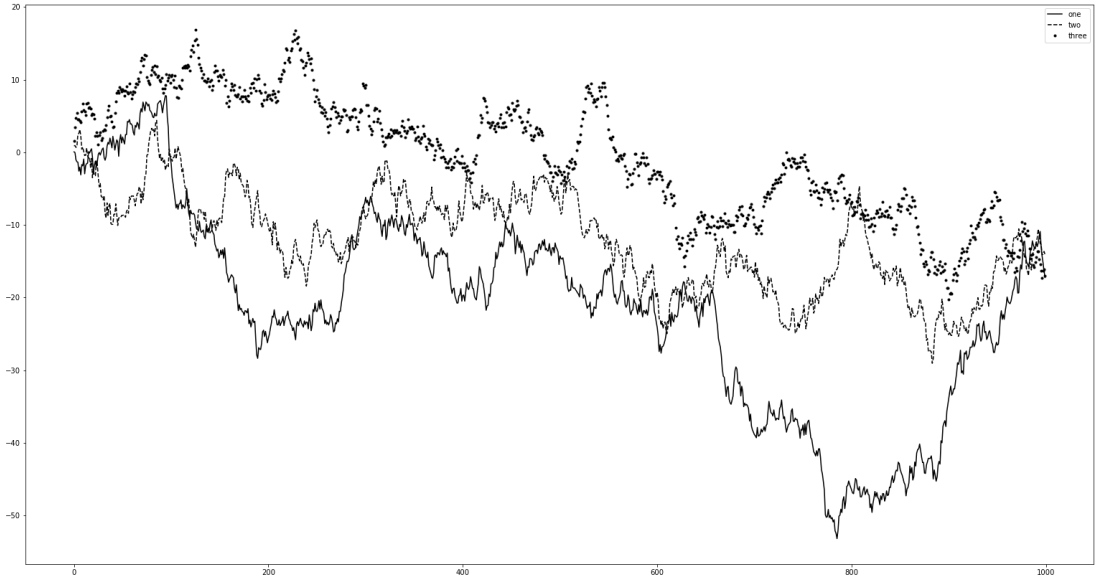






Ticks, Labels, and Legends

- from numpy.random import randn
- fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)
- ax.plot(randn(1000).cumsum(), 'k', label='one')
- ax.plot(randn(1000).cumsum(), 'k--', label='two')
- ax.plot(randn(1000).cumsum(), 'k.', label='three')
- ax.legend(loc='best')



Annotations and Drawing on a Subplot

		SPX
1		
2	1990.2.1 0:00	328.79
3	1990.2.2 0:00	330.92
4	1990.2.5 0:00	331.85
5	1990.2.6 0:00	329.66
6	1990.2.7 0:00	333.75
7	1990.2.8 0:00	332.96
8	1990.2.9 0:00	333.62
9	1990.2.12 0:00	330.08
10	1990.2.13 0:00	331.02

- from datetime import datetime
- fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)
- data = pd.read_csv('spx.csv', index_col=0, parse_dates=True); data.head()

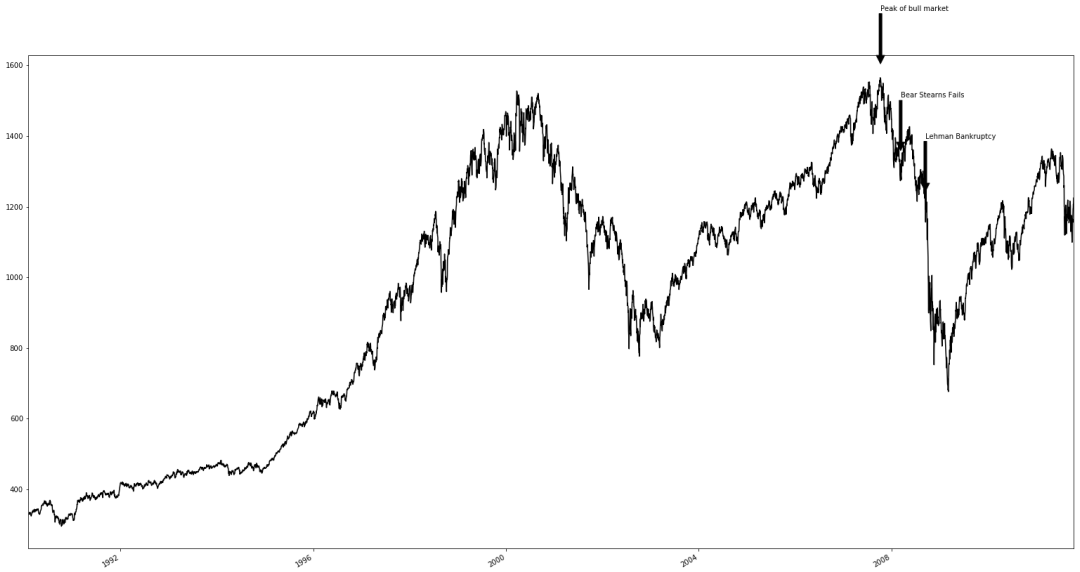
```
                SPX
1990-02-01   328.79
1990-02-02   330.92
1990-02-05   331.85
1990-02-06   329.66
1990-02-07   333.75
```

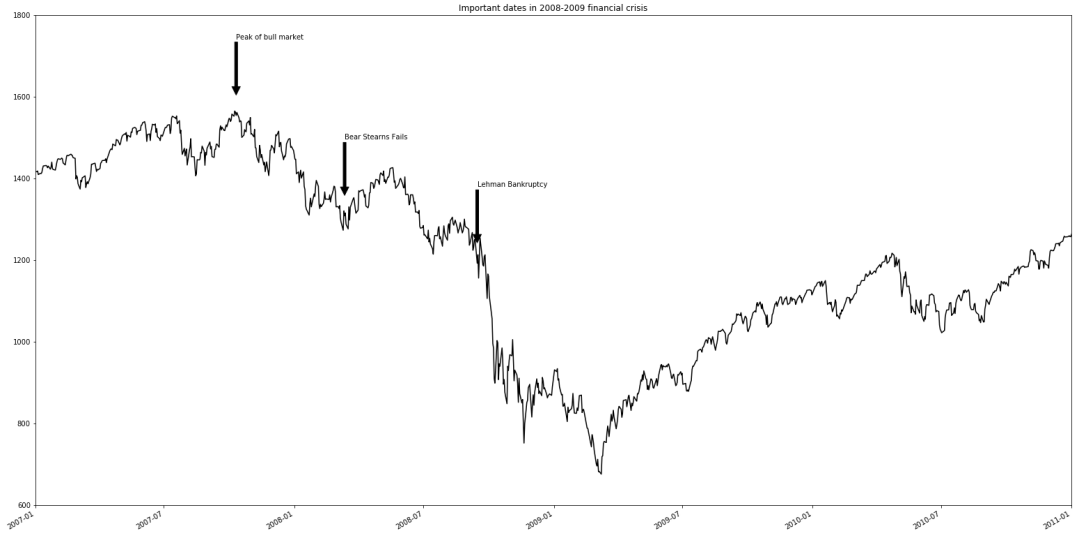
- spx = data['SPX']
- spx.plot(ax=ax, style='k-')
- crisis_data = [(datetime(2007, 10, 11), 'Peak of bull market'), (datetime(2008, 3, 12), 'Bear Stearns Fails'), (datetime(2008, 9, 15), 'Lehman Bankruptcy')]

Annotations and Drawing on a Subplot

- for date, label in crisis_data:
  - ax.annotate(label, xy=(date, spx.asof(date) + 50), xytext=(date, spx.asof(date) + 200), arrowprops=dict(facecolor='black')))
- ax.set_xlim(['1/1/2007', '1/1/2011'])
- ax.set_ylim([600, 1800])
- ax.set_title('Important dates in 2008-2009 financial crisis')

Annotations and Drawing on a Subplot

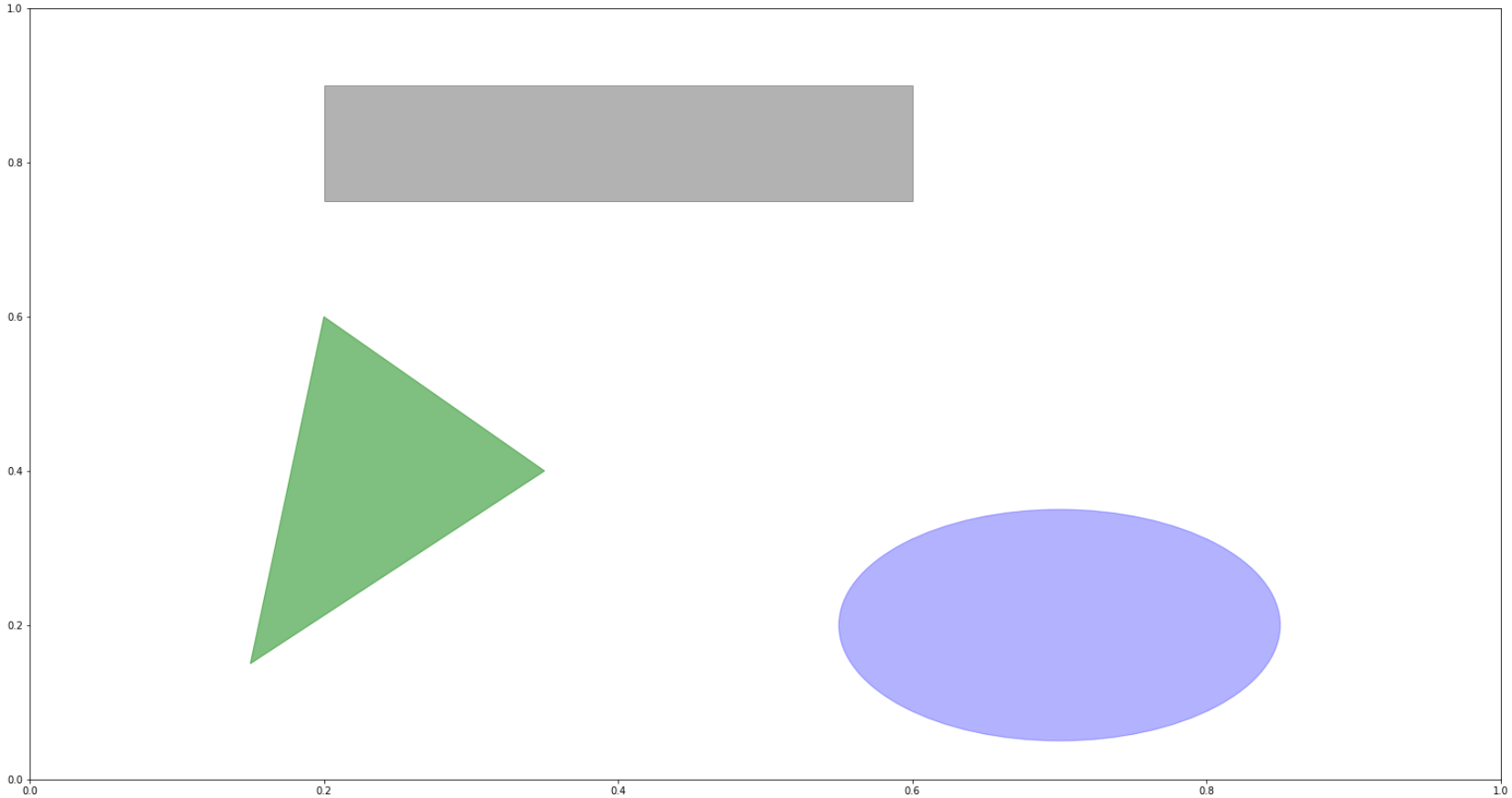




Annotations and Drawing on a Subplot

- Drawing shapes requires some more care. matplotlib has objects that represent many common shapes, referred to as **patches**
- To add a shape to a plot, you create the patch object **shp** and add it to a subplot by calling ax.add_patch(shp)
- fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)
- rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)
- circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)
- pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]], color='g', alpha=0.5)
- ax.add_patch(rect)
- ax.add_patch(circ)
- ax.add_patch(pgon)

Annotations and Drawing on a Subplot



Saving Plots to File

- plt.savefig('figpath.png', dpi=400, bbox_inches='tight')

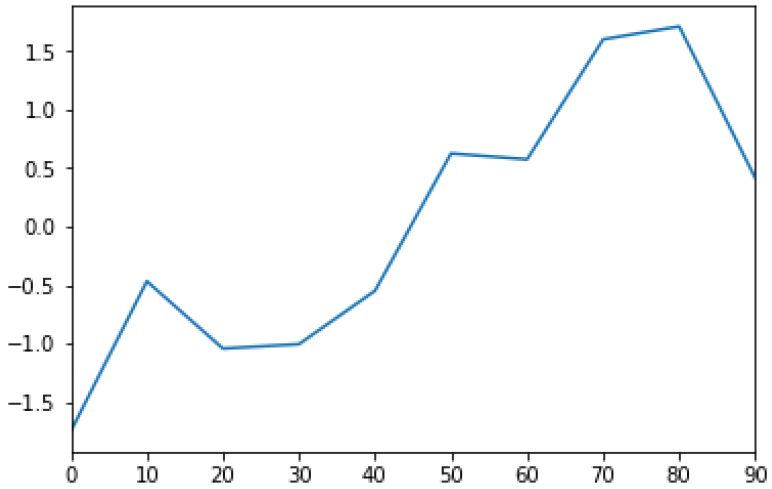
Table 8-2. Figure.savefig options

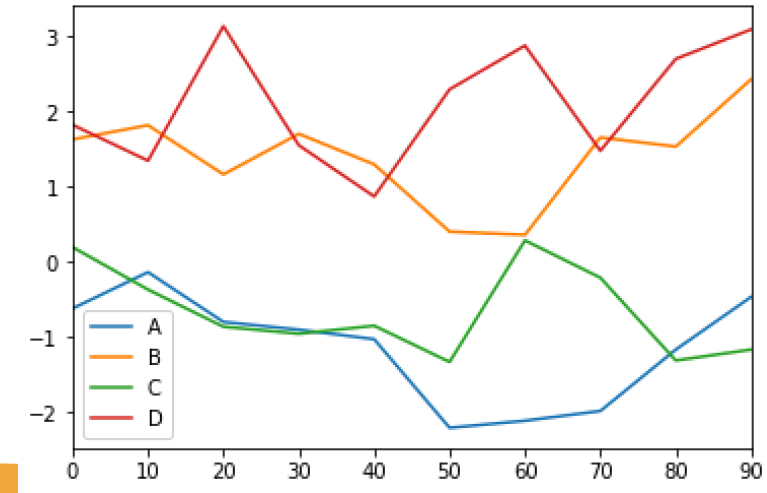
Argument	Description
`fname`	String containing a filepath or a Python file-like object. The figure format is inferred from the file extension, e.g. `.pdf` for PDF or `.png` for PNG.
`dpi`	The figure resolution in dots per inch; defaults to 100 out of the box but can be configured
`facecolor, edgecolor`	The color of the figure background outside of the subplots. `'w'` (white), by default
`format`	The explicit file format to use (`'png', 'pdf', 'svg', 'ps', 'eps', ...`)
`bbox_inches`	The portion of the figure to save. If `'tight'` is passed, will attempt to trim the empty space around the figure

PLOTTING WITH PANDAS AND SEABORN

Line Plots

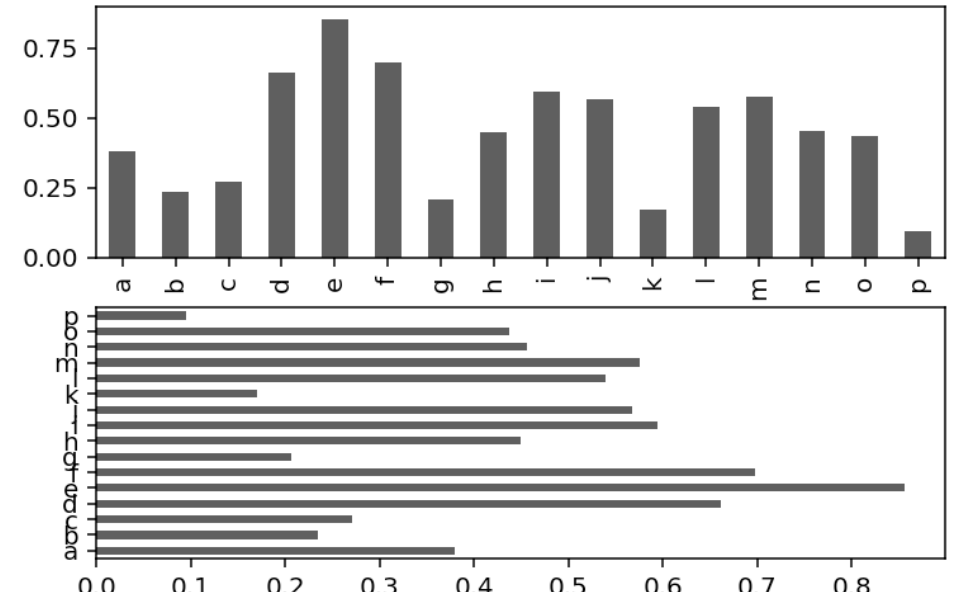
- s = pd.Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10)); s.plot()
- df = pd.DataFrame(np.random.randn(10, 4).cumsum(0), columns=['A', 'B', 'C', 'D'], index=np.arange(0, 100, 10)); df.plot()

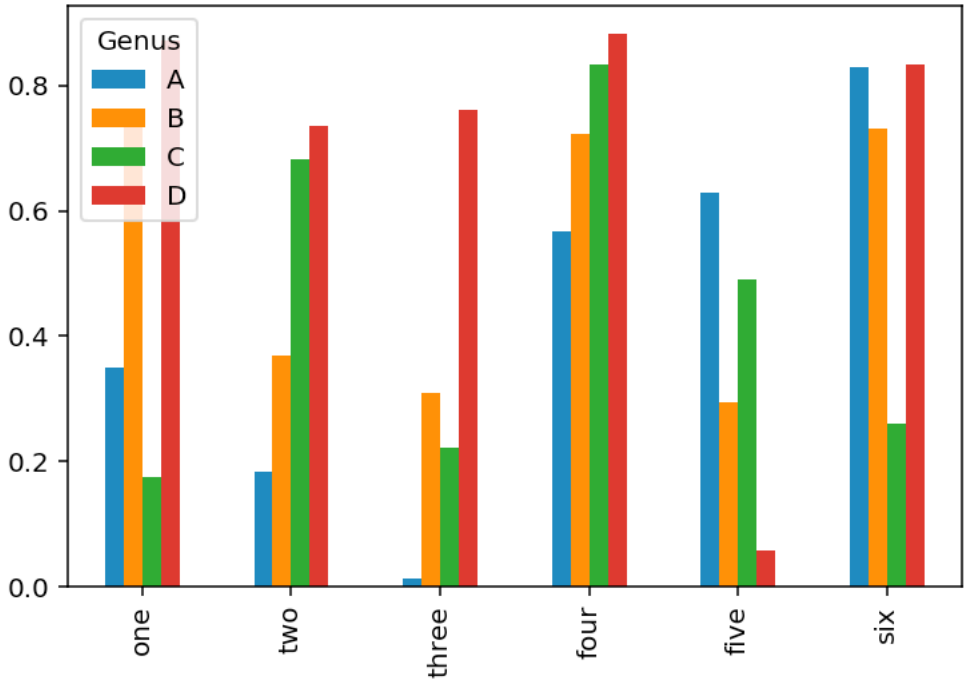




Bar Plots

- fig, axes = plt.subplots(2, 1)
- data = pd.Series(np.random.rand(16), index=list('abcdefghijklmnop'))
- data.plot(kind='bar', ax=axes[0], color='k', alpha=0.7)
- data.plot(kind='barh', ax=axes[1], color='k', alpha=0.7)
- df = pd.DataFrame(np.random.rand(6, 4), columns=pd.Index(['A', 'B', 'C', 'D'], name = 'Genus'), index=['one', 'two', 'three', 'four', 'five', 'six']); df.plot.bar()





Bar Plots

- tips = pd.read_csv('tips.csv'); tips[-5:]
- party_counts = pd.crosstab(tips['day'], tips['size']); party_counts
- party_counts = party_counts.loc[:, 2:5] ; party_counts
- party_pcts = party_counts.div(party_counts.sum(1), axis=0); party_pcts

```
     total_bill   tip smoker   day    time  size
239       29.03  5.92     No   Sat  Dinner     3
240       27.18  2.00    Yes   Sat  Dinner     2
241       22.67  2.00    Yes   Sat  Dinner     2
242       17.82  1.75     No   Sat  Dinner     2
243       18.78  3.00     No  Thur  Dinner     2
```

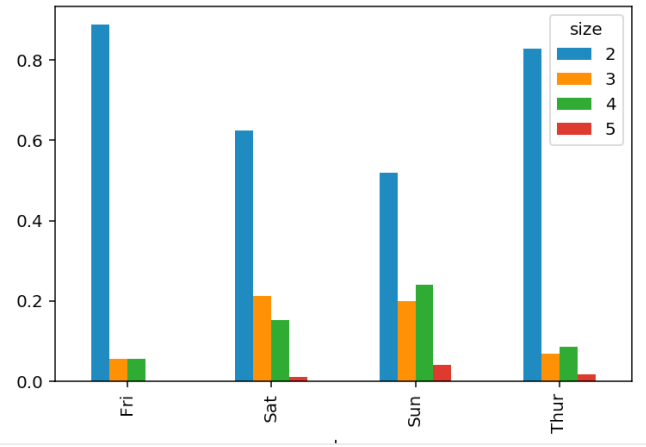
```
size   1   2   3   4  5  6
day
Fri    1  16   1   1  0  0
Sat    2  53  18  13  1  0
Sun    0  39  15  18  3  1
Thur   1  48   4   5  1  3
```

```
size   2   3   4  5
day
Fri   16   1   1  0
Sat   53  18  13  1
Sun   39  15  18  3
Thur  48   4   5  1
```

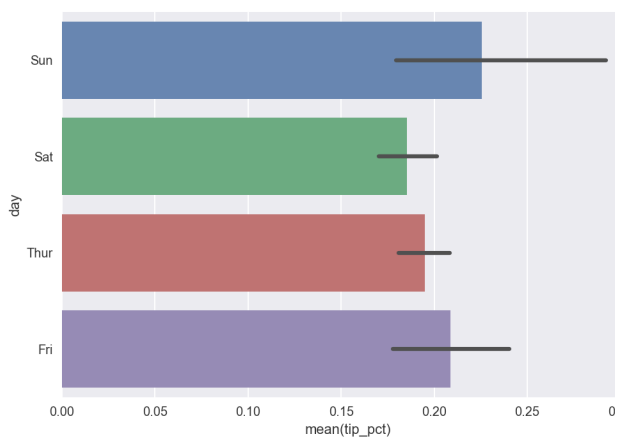
```
size         2         3         4         5
day
Fri   0.888889  0.055556  0.055556  0.000000
Sat   0.623529  0.211765  0.152941  0.011765
Sun   0.520000  0.200000  0.240000  0.040000
Thur  0.827586  0.068966  0.086207  0.017241
```

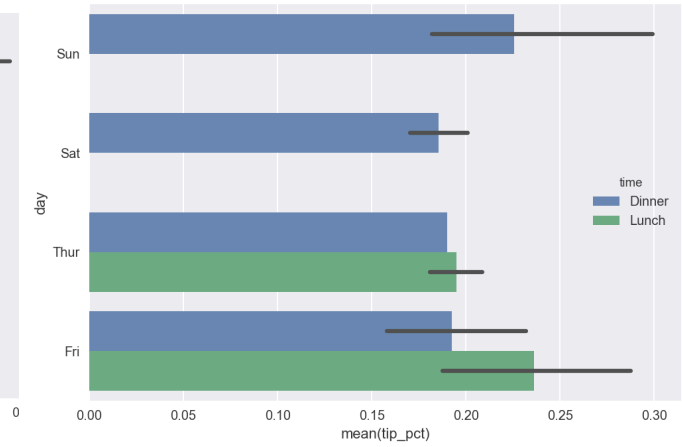
Bar Plots

- party_pcts.plot.bar()
- import seaborn as sns
- tips['tip_pct'] = tips['tip']/(tips['total_bill']-tips['tip']); tips.head()
- sns.barplot(x='tip_pct', y='day', data=tips, orient='h')
- sns.barplot(x='tip_pct', y='day', hue='time', data=tips, orient='h')



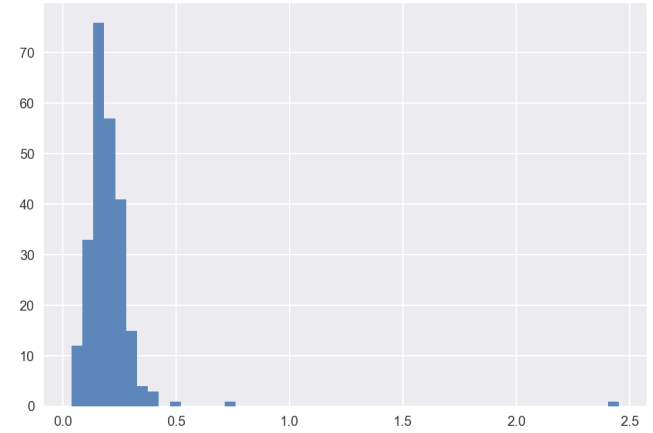
	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.063204
1	10.34	1.66	No	Sun	Dinner	3	0.191244
2	21.01	3.50	No	Sun	Dinner	3	0.199886
3	23.68	3.31	No	Sun	Dinner	2	0.162494
4	24.59	3.61	No	Sun	Dinner	4	0.172069

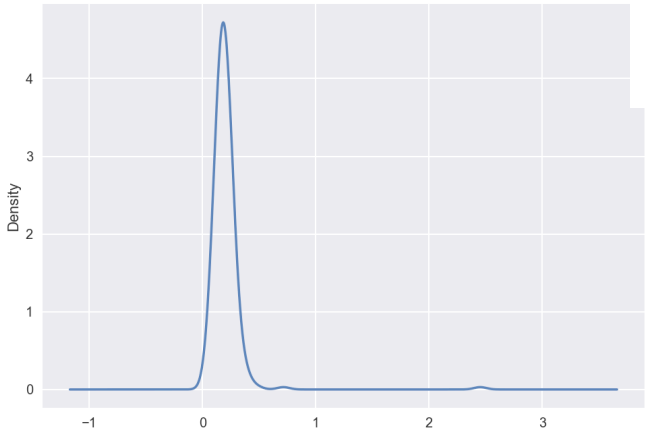


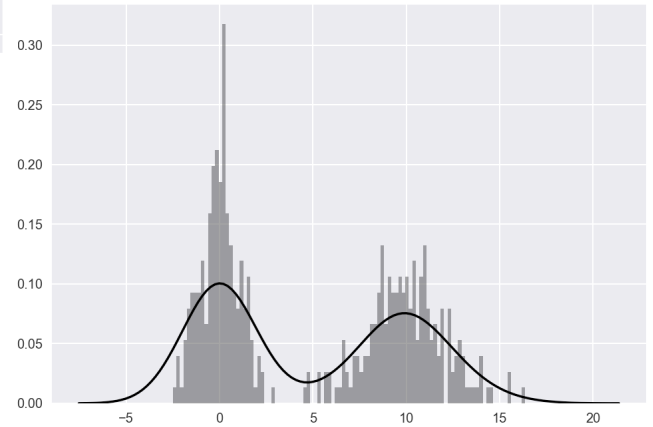


Histograms and Density Plots

- tips['tip_pct'].hist(bins=50)
- tips['tip_pct'].plot.density()
- comp1 = np.random.normal(0, 1, size=200) # N(0, 1)
- comp2 = np.random.normal(10, 2, size=200) # N(10, 4)
- values = pd.Series(np.concatenate([comp1, comp2]))
- sns.distplot(values, bins=100, color='k')



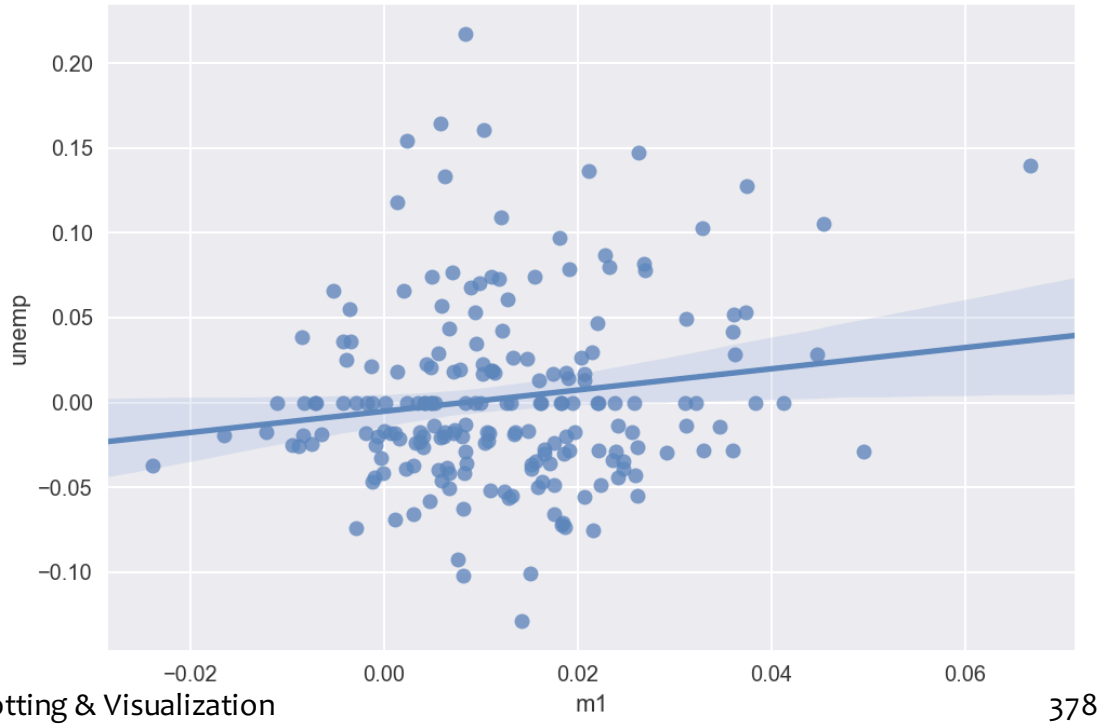




Scatter or Point Plots

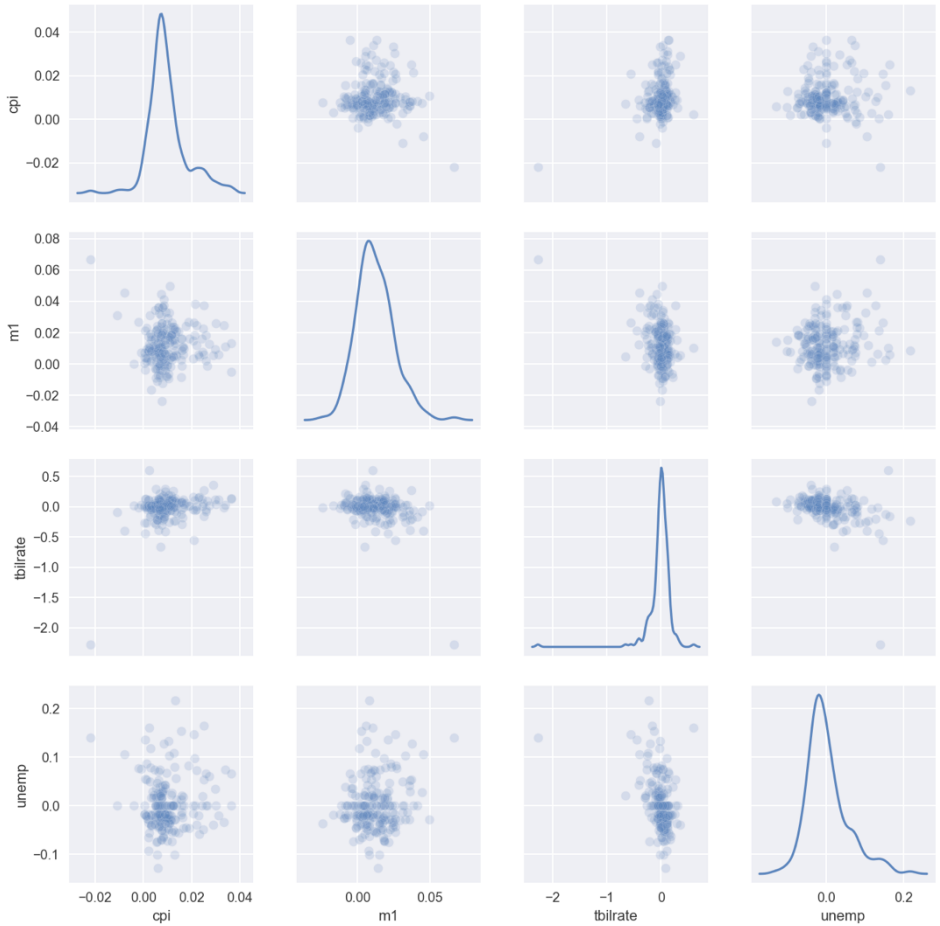
- macro = pd.read_csv('macrodata.csv')
- data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]
- trans_data = np.log(data).diff().dropna()
- trans_data[-5:]
- sns.regplot('m1', 'unemp', data=trans_data)

	cpi	m1	tbilrate	unemp
198	-0.007904	0.045361	-0.396881	0.105361
199	-0.021979	0.066753	-2.277267	0.139762
200	0.002340	0.010286	0.606136	0.160343
201	0.008419	0.037461	-0.200671	0.127339
202	0.008894	0.012202	-0.405465	0.042560



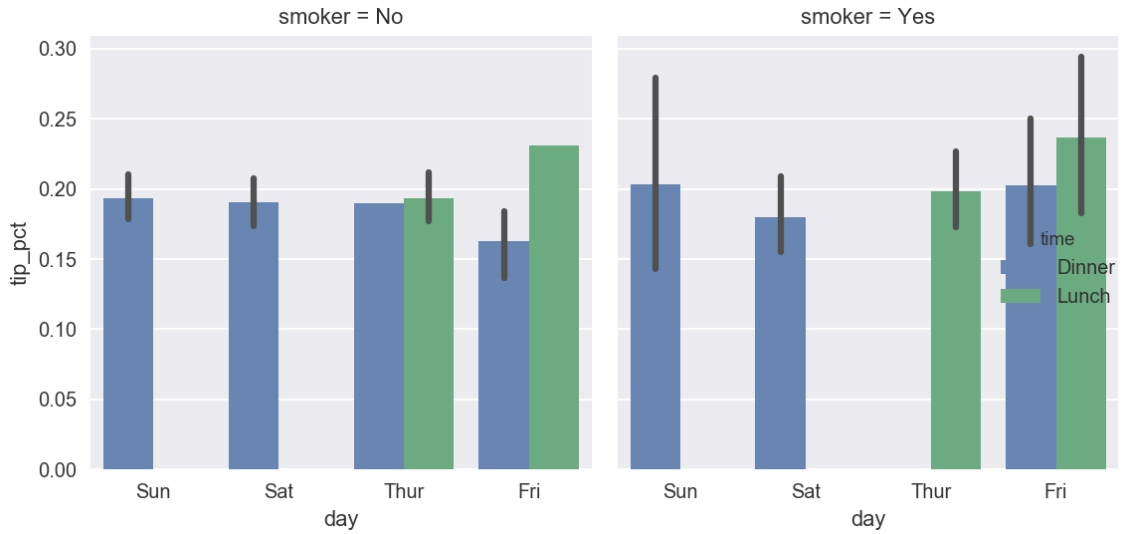
Scatter or Point Plots

- sns.pairplot(trans_data, diag_kind='kde', plot_kws={'alpha': 0.2})



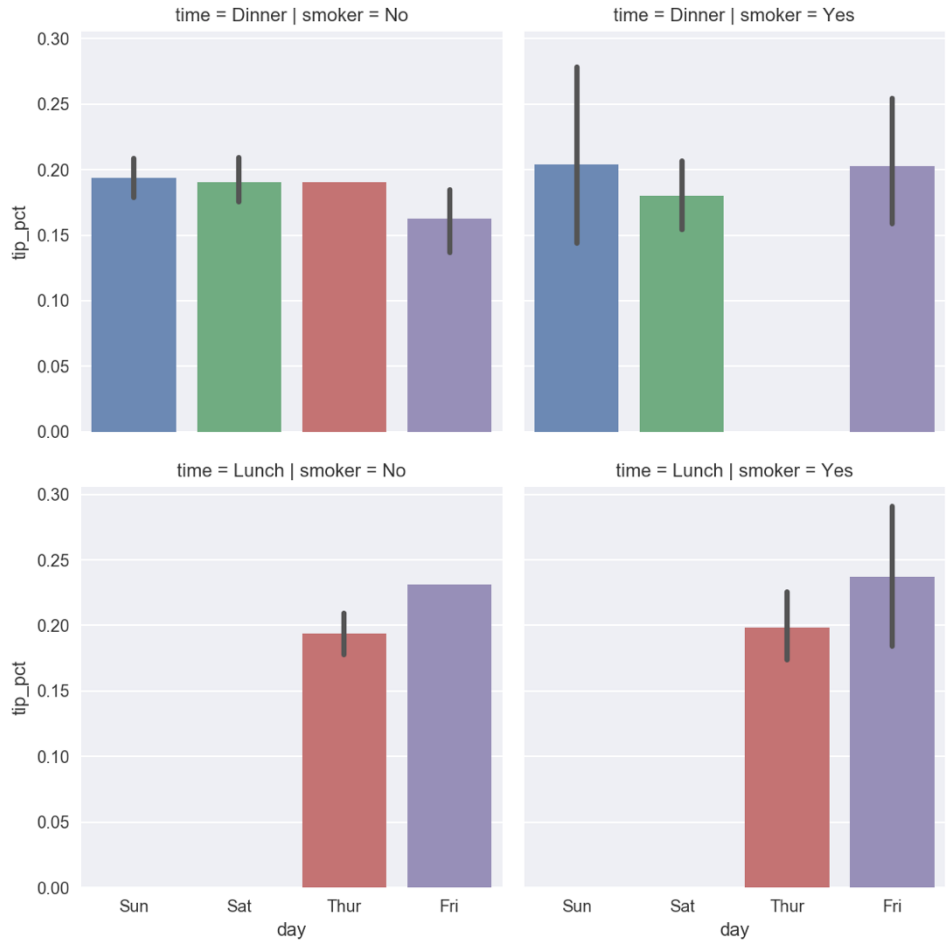
Facet Grids and Categorical Data

- sns.factorplot(x='day', y='tip_pct', hue='time', col='smoker', kind='bar', data=tips[tips.tip_pct<1])



Facet Grids and Categorical Data

- sns.factorplot(x='day', y='tip_pct', row='time', col='smoker', kind='bar', data=tips[tips.tip_pct<1])



Week 14:
Data Aggregation and Group perations

OUTLINE

GroupBy Mechanics

Data Aggregation

Apply: General split-apply-combine

Pivot Tables and Cross-Tabulation

GROUPBY MECHANICS

GroupBy Mechanics

- Each grouping key can take many forms, and the keys do not have to be all of the same type
  - A list or array of values that is the same length as the axis being grouped
  - A value indicating a column name in a DataFrame
  - A dict or Series giving a correspondence between the values on the axis being grouped and the group names
  - A function to be invoked on the axis index or the individual labels in the index

GroupBy Mechanics

- df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'], 'key2' : ['one', 'two', 'one', 'two', 'one'], 'data1' : np.random.randn(5), 'data2' : np.random.randn(5)}); df
- grouped = df['data1'].groupby(df['key1'])
- grouped.mean()
- means = df['data1'].groupby([df['key1'], df['key2']]).mean(); means

```
      data1     data2 key1 key2
0 -0.441374 -1.285534    a  one
1  1.082282  0.850204    a  two
2  1.074037  1.805361    b  one
3 -0.377948  1.020249    b  two
4  0.374822 -0.660006    a  one
```

```
key1
a    0.338577
b    0.348045
Name: data1, dtype: float64
```

```
key1  key2
a     one    -0.033276
      two     1.082282
b     one     1.074037
      two    -0.377948
Name: data1, dtype: float64
```

GroupBy Mechanics

- states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])
- years = np.array([2005, 2005, 2006, 2005, 2006])
- df['data1'].groupby([states, years]).mean()
- df.groupby('key1').mean()
- df.groupby(['key1', 'key2']).mean()

```
California  2005    1.082282
            2006    1.074037
Ohio        2005   -0.409661
            2006    0.374822
Name: data1, dtype: float64
```

```
          data1     data2
key1
a      0.338577 -0.365112
b      0.348045  1.412805
```

```
               data1     data2
key1 key2
a    one   -0.033276 -0.972770
     two    1.082282  0.850204
b    one    1.074037  1.805361
     two   -0.377948  1.020249
```

Iterating Over Groups

- for name, group in df.groupby('key1'):
  - print(name)
  - print(group)
- for (k1, k2), group in df.groupby(['key1', 'key2']):
  - print(k1, k2)
  - print(group)

```
a
      data1     data2 key1 key2
0 -0.441374 -1.285534    a  one
1  1.082282  0.850204    a  two
4  0.374822 -0.660006    a  one
b
      data1     data2 key1 key2
2  1.074037  1.805361    b  one
3 -0.377948  1.020249    b  two
```

```
a one
      data1     data2 key1 key2
0 -0.441374 -1.285534    a  one
4  0.374822 -0.660006    a  one
a two
      data1     data2 key1 key2
1  1.082282  0.850204    a  two
b one
      data1     data2 key1 key2
2  1.074037  1.805361    b  one
b two
      data1     data2 key1 key2
3 -0.377948  1.020249    b  two
```

Iterating Over Groups

- df.dtypes
- grouped = df.groupby(df.dtypes, axis=1)
- for name, group in grouped:
  - print(name)
  - print(group)

```
data1    float64
data2    float64
key1      object
key2      object
dtype: object
```

```
float64
       data1     data2
0 -0.441374 -1.285534
1  1.082282  0.850204
2  1.074037  1.805361
3 -0.377948  1.020249
4  0.374822 -0.660006
object
  key1 key2
0    a  one
1    a  two
2    b  one
3    b  two
4    a  one
```

Selecting a Column or Subset of Columns

- d_grouped = df.groupby(['key1', 'key2'])[['data2']].mean();
  d_grouped
- s_grouped = df.groupby(['key1', 'key2'])['data2'].mean();
  s_grouped

```
                 data2
key1 key2
a    one   -0.972770
     two    0.850204
b    one    1.805361
     two    1.020249
```

```
key1  key2
a     one    -0.972770
      two     0.850204
b     one     1.805361
      two     1.020249
Name: data2, dtype: float64
```

Grouping with Dicts and Series

- people = pd.DataFrame(np.random.randn(5, 5), columns=['a', 'b', 'c', 'd', 'e'], index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis']); people
- people.iloc[2:3, [1, 2]] = np.nan; people
- mapping = {'a': 'red', 'b': 'red', 'c': 'blue', 'd': 'blue', 'e': 'red', 'f' : 'orange'}
- by_column = people.groupby(mapping, axis=1)

	a	b	c	d	e
Joe	0.717879	-0.786513	1.177543	0.047897	-0.760276
Steve	-0.578792	-0.189619	0.027822	-0.386705	0.299336
Wes	-0.401558	1.004596	-1.220623	0.329282	0.484441
Jim	1.568370	-0.341319	0.059881	0.179860	1.784991
Travis	-0.608088	0.277185	-0.117172	0.066928	0.129202

	a	b	c	d	e
Joe	0.717879	-0.786513	1.177543	0.047897	-0.760276
Steve	-0.578792	-0.189619	0.027822	-0.386705	0.299336
Wes	-0.401558	NaN	NaN	0.329282	0.484441
Jim	1.568370	-0.341319	0.059881	0.179860	1.784991
Travis	-0.608088	0.277185	-0.117172	0.066928	0.129202

Grouping with Dicts and Series

- for name, group in by_column:
  - print(name)
  - print(group)
- by_column.sum()

```
blue
                c         d
Joe      1.177543  0.047897
Steve    0.027822 -0.386705
Wes           NaN  0.329282
Jim      0.059881  0.179860
Travis -0.117172  0.066928
red
                a         b         e
Joe      0.717879 -0.786513 -0.760276
Steve   -0.578792 -0.189619  0.299336
Wes     -0.401558       NaN  0.484441
Jim      1.568370 -0.341319  1.784991
Travis -0.608088  0.277185  0.129202
```

```
            blue       red
Joe     1.225439 -0.828911
Steve  -0.358883 -0.469075
Wes     0.329282  0.082883
Jim     0.239741  3.012041
Travis -0.050244 -0.201702
```

Grouping with Dicts and Series

- map_series = pd.Series(mapping)
- map_series
- s_grouped = people.groupby(map_series, axis=1)
- for name, group in s_grouped:
  - print(name)
  - print(group)
- s_grouped.count()

```
a       red
b       red
c      blue
d      blue
e       red
f    orange
dtype: object
```

```
blue
              c         d
Joe    1.177543  0.047897
Steve  0.027822 -0.386705
Wes         NaN  0.329282
Jim    0.059881  0.179860
Travis -0.117172  0.066928
red
              a         b         e
Joe    0.717879 -0.786513 -0.760276
Steve -0.578792 -0.189619  0.299336
Wes   -0.401558       NaN  0.484441
Jim    1.568370 -0.341319  1.784991
Travis -0.608088  0.277185  0.129202
```

```
        blue  red
Joe        2    3
Steve      2    3
Wes        1    2
Jim        2    3
Travis     2    3
```

Grouping with Functions

- f_grouped = people.groupby(len)
- for name, group in f_grouped:
  - print(name)
  - print(group)
- f_grouped.sum()

```
3
          a         b         c         d         e
Joe  0.717879 -0.786513  1.177543  0.047897 -0.760276
Wes -0.401558       NaN       NaN  0.329282  0.484441
Jim  1.568370 -0.341319  0.059881  0.179860  1.784991
5
               a         b         c         d         e
Steve -0.578792 -0.189619  0.027822 -0.386705  0.299336
6
                 a         b         c         d         e
Travis -0.608088  0.277185 -0.117172  0.066928  0.129202
```

```
          a         b         c         d         e
3  1.884690 -1.127833  1.237424  0.557038  1.509155
5 -0.578792 -0.189619  0.027822 -0.386705  0.299336
6 -0.608088  0.277185 -0.117172  0.066928  0.129202
```

Grouping with Functions

- Mixing functions with arrays, dicts, or Series is not a problem as everything gets converted to arrays internally
- key_list = ['one', 'one', 'one', 'two', 'two']
- m_grouped = people.groupby([len, key_list])
- for (k1, k2), group in m_grouped:
  - print(k1, k2)
  - print(group)
- m_grouped.min()

```
3 one
          a         b         c         d         e
Joe  0.717879 -0.786513  1.177543  0.047897 -0.760276
Wes -0.401558       NaN       NaN  0.329282  0.484441
3 two
          a         b         c         d         e
Jim  1.56837 -0.341319  0.059881   0.17986  1.784991
5 one
           a         b         c         d         e
Steve -0.578792 -0.189619  0.027822 -0.386705  0.299336
6 two
            a         b         c         d         e
Travis -0.608088  0.277185 -0.117172  0.066928  0.129202
```

```
                a         b         c         d         e
3 one -0.401558 -0.786513  1.177543  0.047897 -0.760276
  two  1.568370 -0.341319  0.059881  0.179860  1.784991
5 one -0.578792 -0.189619  0.027822 -0.386705  0.299336
6 two -0.608088  0.277185 -0.117172  0.066928  0.129202
```

Grouping by Index Levels

- columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP', 'JP'], [1, 3, 5, 1, 3]], names=['cty', 'tenor']); columns
- Out[]: MultiIndex(levels=[['JP', 'US'], [1, 3, 5]], labels=[[1, 1, 1, 0, 0], [0, 1, 2, 0, 1]], names=['cty', 'tenor'])
- hier_df = pd.DataFrame(np.random.randn(4, 5), columns=columns); hier_df

cty	US			JP	
tenor	1	3	5	1	3
0	0.140175	-0.702217	0.584119	-1.034711	-0.056024
1	-0.551409	0.312316	-1.618980	0.075199	-1.066202
2	0.580363	-0.624540	0.323682	1.613998	-0.405646
3	-1.104169	-1.781861	-0.191496	-1.479188	-0.623011

Grouping by Index Levels

- lc_grouped = hier_df.groupby(level='cty', axis=1)
- lt_grouped = hier_df.groupby(level='tenor', axis=1)
- for name, group in lc_grouped:
  - print(name)
  - print(group)
- for name, group in lt_grouped:
  - print(name)
  - print(group)
- lc_grouped.count()
- lt_grouped.count()

```
JP
cty          JP
tenor         1         3
0     -1.034711 -0.056024
1      0.075199 -1.066202
2      1.613998 -0.405646
3     -1.479188 -0.623011
US
cty          US
tenor         1         3         5
0      0.140175 -0.702217  0.584119
1     -0.551409  0.312316 -1.618980
2      0.580363 -0.624540  0.323682
3     -1.104169 -1.781861 -0.191496
```

```
1
cty          US        JP
tenor         1         1
0      0.140175 -1.034711
1     -0.551409  0.075199
2      0.580363  1.613998
3     -1.104169 -1.479188
3
cty          US        JP
tenor         3         3
0     -0.702217 -0.056024
1      0.312316 -1.066202
2     -0.624540 -0.405646
3     -1.781861 -0.623011
5
cty          US
tenor         5
0      0.584119
1     -1.618980
2      0.323682
3     -0.191496
```

cty	JP	US
0	2	3
1	2	3
2	2	3
3	2	3

tenor	1	3	5
0	2	2	1
1	2	2	1
2	2	2	1
3	2	2	1

DATA AGGREGATION

Data Aggregation

- df
- grouped = df.groupby('key1')
- def peak_to_peak(arr):
  - return arr.max() - arr.min()
- grouped.agg(peak_to_peak)
- grouped.describe()

```
      data1       data2 key1 key2
0 -0.441374 -1.285534     a  one
1  1.082282  0.850204     a  two
2  1.074037  1.805361     b  one
3 -0.377948  1.020249     b  two
4  0.374822 -0.660006     a  one
```

```
          data1     data2
key1
a      1.523656  2.135738
b      1.451984  0.785112
```

```
     data1                                                                \
     count      mean       std       min       25%       50%       75%
key1
a      3.0  0.338577  0.762474 -0.441374 -0.033276  0.374822  0.728552
b      2.0  0.348045  1.026708 -0.377948 -0.014952  0.348045  0.711041

             data2                                                        \
         max count      mean       std       min       25%       50%
key1
a      1.082282   3.0 -0.365112  1.097983 -1.285534 -0.972770 -0.660006
b      1.074037   2.0  1.412805  0.555158  1.020249  1.216527  1.412805


          75%       max
key1
a     0.095099  0.850204
b     1.609083  1.805361
```

Column-wise and Multiple Function Application

- tips = pd.read_csv('tips.csv')
- tips['tip_pct'] = tips['tip'] / tips['total_bill']; tips.head()
- grouped = tips.groupby(['day', 'smoker'])
- grouped_pct = grouped['tip_pct']
- grouped_pct.agg('mean')

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808

```
day   smoker
Fri   No        0.151650
      Yes       0.174783
Sat   No        0.158048
      Yes       0.147906
Sun   No        0.160113
      Yes       0.187250
Thur  No        0.160298
      Yes       0.163863
Name: tip_pct, dtype: float64
```

Column-wise and Multiple Function Application

- grouped_pct.agg([('foo', 'mean'), ('bar', np.std)])
- functions = ['count', 'mean', 'max']
- result = grouped['tip_pct', 'total_bill'].agg(functions); result

		foo	bar
day	smoker		
Fri	No	0.151650	0.028123
	Yes	0.174783	0.051293
Sat	No	0.158048	0.039767
	Yes	0.147906	0.061375
Sun	No	0.160113	0.042347
	Yes	0.187250	0.154134
Thur	No	0.160298	0.038774
	Yes	0.163863	0.039389

		tip_pct			total_bill		
		count	mean	max	count	mean	max
day	smoker						
Fri	No	4	0.151650	0.187735	4	18.420000	22.75
	Yes	15	0.174783	0.263480	15	16.813333	40.17
Sat	No	45	0.158048	0.291990	45	19.661778	48.33
	Yes	42	0.147906	0.325733	42	21.276667	50.81
Sun	No	57	0.160113	0.252672	57	20.506667	48.17
	Yes	19	0.187250	0.710345	19	24.120000	45.35
Thur	No	45	0.160298	0.266312	45	17.113111	41.19
	Yes	17	0.163863	0.241255	17	19.190588	43.11

Column-wise and Multiple Function Application

- ftuples = [('Durchschnitt', 'mean'), ('Abweichung', np.var)]
- grouped['tip_pct', 'total_bill'].agg(ftuples)
- grouped.agg({'tip' : np.max, 'size' : 'sum'})
- grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'], 'size' : 'sum'})

		tip_pct		total_bill	
		Durchschnitt	Abweichung	Durchschnitt	Abweichung
day	smoker				
Fri	No	0.151650	0.000791	18.420000	25.596333
	Yes	0.174783	0.002631	16.813333	82.562438
Sat	No	0.158048	0.001581	19.661778	79.908965
	Yes	0.147906	0.003767	21.276667	101.387535
Sun	No	0.160113	0.001793	20.506667	66.099980
	Yes	0.187250	0.023757	24.120000	109.046044
Thur	No	0.160298	0.001503	17.113111	59.625081
	Yes	0.163863	0.001551	19.190588	69.808518

		tip	size
day	smoker		
Fri	No	3.50	9
	Yes	4.73	31
Sat	No	9.00	115
	Yes	10.00	104
Sun	No	6.00	167
	Yes	6.50	49
Thur	No	6.70	112
	Yes	5.00	40

		tip_pct				size
		min	max	mean	std	sum
day	smoker					
Fri	No	0.120385	0.187735	0.151650	0.028123	9
	Yes	0.103555	0.263480	0.174783	0.051293	31
Sat	No	0.056797	0.291990	0.158048	0.039767	115
	Yes	0.035638	0.325733	0.147906	0.061375	104
Sun	No	0.059447	0.252672	0.160113	0.042347	167
	Yes	0.065660	0.710345	0.187250	0.154134	49
Thur	No	0.072961	0.266312	0.160298	0.038774	112
	Yes	0.090014	0.241255	0.163863	0.039389	40

Returning Aggregated Data Without Row Indexes

- wr_grouped = tips.groupby(['day', 'smoker'], as_index=False)
- wr_grouped.mean()

	day	smoker	total_bill	tip	size	tip_pct
0	Fri	No	18.420000	2.812500	2.250000	0.151650
1	Fri	Yes	16.813333	2.714000	2.066667	0.174783
2	Sat	No	19.661778	3.102889	2.555556	0.158048
3	Sat	Yes	21.276667	2.875476	2.476190	0.147906
4	Sun	No	20.506667	3.167895	2.929825	0.160113
5	Sun	Yes	24.120000	3.516842	2.578947	0.187250
6	Thur	No	17.113111	2.673778	2.488889	0.160298
7	Thur	Yes	19.190588	3.030000	2.352941	0.163863

APPLY: GENERAL SPLIT-APPLY-COMBINE

Apply: General split-apply-combine

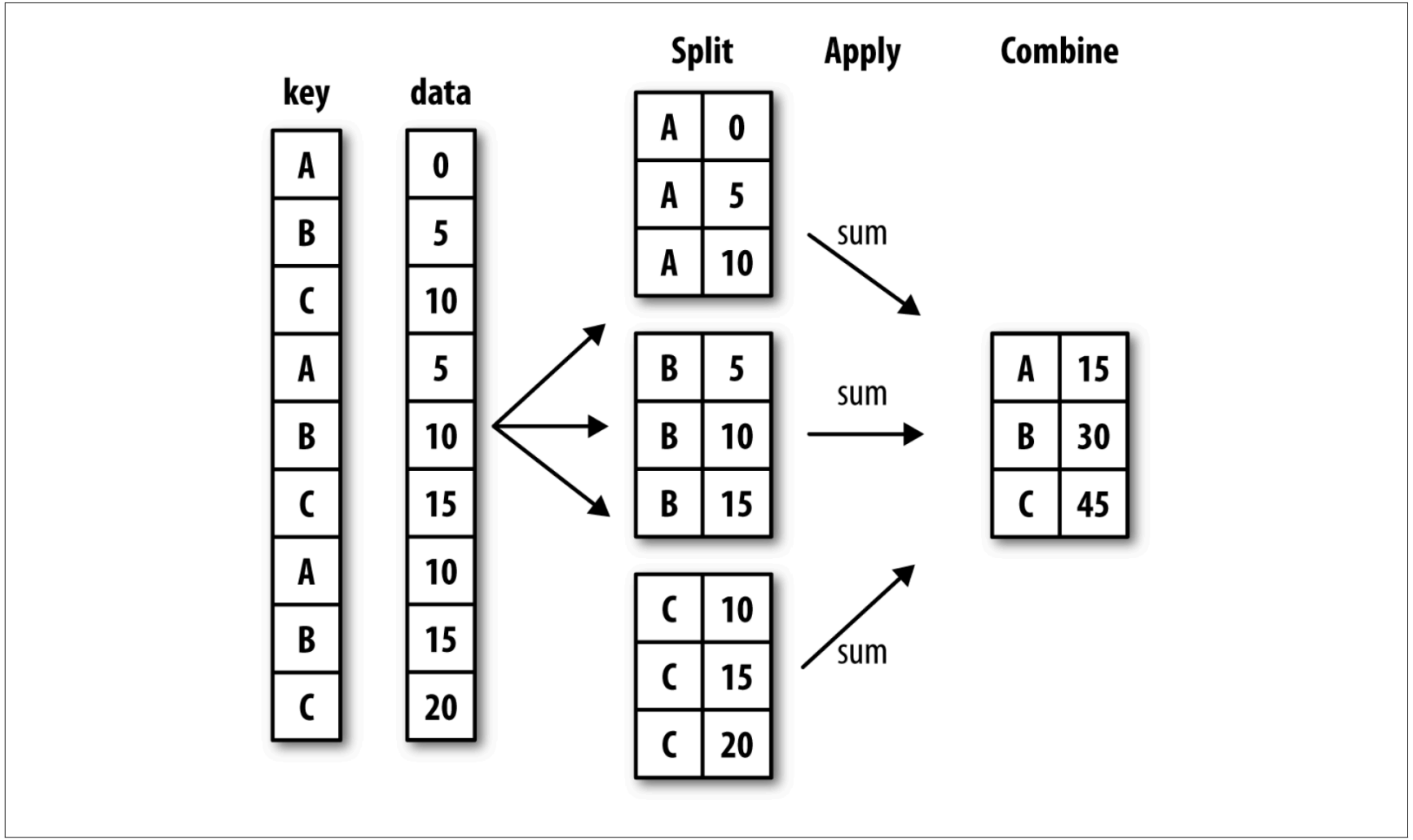


Figure 9-1. Illustration of a group aggregation

Apply: General split-apply-combine

- Suppose you wanted to select the top five tip_pct values by group.
- def top(df, n=5, column='tip_pct'):
  - return df.sort_values(by=column)[-n:]
- top(tips, n=6)
- tips.groupby('smoker').apply(top)
  - The result has a hierarchical index whose inner level contains index values from the original DataFrame

	total_bill	tip	smoker	day	time	size	tip_pct
109	14.31	4.00	Yes	Sat	Dinner	2	0.279525
183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
232	11.61	3.39	No	Sat	Dinner	2	0.291990
67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
172	7.25	5.15	Yes	Sun	Dinner	2	0.710345

		total_bill	tip	smoker	day	time	size	tip_pct
smoker								
No	88	24.71	5.85	No	Thur	Lunch	2	0.236746
	185	20.69	5.00	No	Sun	Dinner	5	0.241663
	51	10.29	2.60	No	Sun	Dinner	2	0.252672
	149	7.51	2.00	No	Thur	Lunch	2	0.266312
	232	11.61	3.39	No	Sat	Dinner	2	0.291990
Yes	109	14.31	4.00	Yes	Sat	Dinner	2	0.279525
	183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
	67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
	178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
	172	7.25	5.15	Yes	Sun	Dinner	2	0.710345

Apply: General split-apply-combine

- tips.groupby(['smoker', 'day']).apply(top, n=1, column='total_bill')
- result = tips.groupby('smoker')['tip_pct'].describe(); result;
- result.unstack('smoker')

```
                  total_bill    tip smoker   day    time  size   tip_pct
smoker day
No     Fri  94        22.75   3.25     No   Fri  Dinner     2  0.142857
       Sat  212       48.33   9.00     No   Sat  Dinner     4  0.186220
       Sun  156       48.17   5.00     No   Sun  Dinner     6  0.103799
       Thur 142       41.19   5.00     No  Thur   Lunch     5  0.121389
Yes    Fri  95        40.17   4.73    Yes   Fri  Dinner     4  0.117750
       Sat  170       50.81  10.00    Yes   Sat  Dinner     3  0.196812
       Sun  182       45.35   3.50    Yes   Sun  Dinner     3  0.077178
       Thur 197       43.11   5.00    Yes  Thur   Lunch     4  0.115982
```

```
        count      mean       std       min       25%       50%       75%  \
smoker
No      151.0  0.159328  0.039910  0.056797  0.136906  0.155625  0.185014
Yes      93.0  0.163196  0.085119  0.035638  0.106771  0.153846  0.195059

             max
smoker
No      0.291990
Yes     0.710345
```

```
        smoker
count   No         151.000000
        Yes         93.000000
mean    No           0.159328
        Yes          0.163196
std     No           0.039910
        Yes          0.085119
min     No           0.056797
        Yes          0.035638
25%     No           0.136906
        Yes          0.106771
50%     No           0.155625
        Yes          0.153846
75%     No           0.185014
        Yes          0.195059
max     No           0.291990
        Yes          0.710345
dtype: float64
```

Suppressing the Group Keys

- The resulting object has a hierarchical index formed from the group keys along with the indexes of each piece of the original object
- You can disable this by passing **group_keys=False** to **groupby**

- tips.groupby('smoker', group_keys=False).apply(top)

	total_bill	tip	smoker	day	time	size	tip_pct
88	24.71	5.85	No	Thur	Lunch	2	0.236746
185	20.69	5.00	No	Sun	Dinner	5	0.241663
51	10.29	2.60	No	Sun	Dinner	2	0.252672
149	7.51	2.00	No	Thur	Lunch	2	0.266312
232	11.61	3.39	No	Sat	Dinner	2	0.291990
109	14.31	4.00	Yes	Sat	Dinner	2	0.279525
183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
172	7.25	5.15	Yes	Sun	Dinner	2	0.710345

Quantile and Bucket Analysis

- frame = pd.DataFrame({'data1': np.random.randn(1000), 'data2':np.random.randn(1000)})
- quantile = pd.cut(frame.data1, 4)
- quantile[:10]

```
0     (-0.108, 1.387]
1     (-0.108, 1.387]
2     (-0.108, 1.387]
3    (-1.604, -0.108]
4    (-1.604, -0.108]
5     (-0.108, 1.387]
6     (-0.108, 1.387]
7    (-3.106, -1.604]
8     (-0.108, 1.387]
9     (-0.108, 1.387]
Name: data1, dtype: category
Categories (4, interval[float64]): [(-3.106, -1.604] < (-1.604, -0.108] < (-0.108, 1.387] < (1.387, 2.883]]
```

Quantile and Bucket Analysis

- def get_stats(group):
  - return {'min': group.min(), 'max': group.max(),
  - 'count': group.count(), 'mean': group.mean()}
- grouped = frame.data2.groupby(quantile)
- grouped. apply(get_stats)
- grouped.apply(get_stats).unstack()

```
data1
(-3.106, -1.604]   count     52.000000
                   max        2.288044
                   mean       0.128885
                   min       -2.035179
(-1.604, -0.108]   count    403.000000
                   max        2.931616
                   mean      -0.045618
                   min       -3.366113
(-0.108, 1.387]    count    469.000000
                   max        2.738414
                   mean       0.025470
                   min       -3.125678
(1.387, 2.883]     count     76.000000
                   max        2.360718
                   mean       0.089382
                   min       -2.351573
Name: data2, dtype: float64
```

	count	max	mean	min
data1				
(-3.106, -1.604]	52.0	2.288044	0.128885	-2.035179
(-1.604, -0.108]	403.0	2.931616	-0.045618	-3.366113
(-0.108, 1.387]	469.0	2.738414	0.025470	-3.125678
(1.387, 2.883]	76.0	2.360718	0.089382	-2.351573

Quantile and Bucket Analysis

- grouping = pd.qcut(frame.data1, 10, labels=False)
- grouped = frame.data2.groupby(grouping)
- grouped.apply(get_stats).unstack()

data1	count	max	mean	min
0	100.0	2.288044	0.030199	-3.366113
1	100.0	2.393390	-0.071833	-2.418605
2	100.0	2.165073	0.012907	-1.817027
3	100.0	2.931616	-0.178389	-2.619439
4	100.0	2.844573	0.091119	-2.230540
5	100.0	2.148813	0.048570	-2.249539
6	100.0	2.042033	0.108614	-2.833230
7	100.0	2.092938	-0.051563	-2.777026
8	100.0	2.738414	-0.037283	-3.125678
9	100.0	2.360718	0.118220	-2.351573

Example: Filling Missing Values with Group-specific Values

- s = pd.Series(np.random.randn(6))
- s[::2] = np.nan; s
- s.mean()
- Out[]: 0.3109951443168812 4
- s.fillna(s.mean())

```
0         NaN
1    2.095592
2         NaN
3   -1.108266
4         NaN
5   -0.054340
dtype: float64
```

```
0    0.310995
1    2.095592
2    0.310995
3   -1.108266
4    0.310995
5   -0.054340
dtype: float64
```

Example: Filling Missing Values with Group-specific Values

- states = ['Ohio', 'New York', 'Vermont', 'Florida', 'Oregon', 'Nevada', 'California', 'Idaho']
- data = pd.Series(np.random.randn(8), index=states); data
- data[['Vermont', 'Nevada', 'Idaho']] = np.n; data

```
Ohio          -0.705493
New York      -0.895493
Vermont        0.205786
Florida       -0.008667
Oregon        -0.170282
Nevada         2.056641
California     0.960208
Idaho          0.687468
dtype: float64
```

```
Ohio          -0.705493
New York      -0.895493
Vermont             NaN
Florida       -0.008667
Oregon        -0.170282
Nevada              NaN
California     0.960208
Idaho               NaN
dtype: float64
```

Example: Filling Missing Values with Group-specific Values

- group_key = ['East'] * 4 + ['West'] * 4
- data.groupby(group_key).mean()
- fill_mean = lambda g: g.fillna(g.mean())
- data.groupby(group_key).apply(fill_mean)

```
East   -0.536551
West    0.394963
dtype: float64
```

```
Ohio          -0.705493
New York      -0.895493
Vermont       -0.536551
Florida       -0.008667
Oregon        -0.170282
Nevada         0.394963
California     0.960208
Idaho          0.394963
dtype: float64
```

Example: Filling Missing Values with Group-specific Values

- fill_values = {'East': 0.5, 'West': -1}
- fill_func = lambda g: g.fillna(fill_values[g.name]) # the groups have a name attribute set internally
- data.groupby(group_key).apply(fill_func)

```
Ohio          -0.705493
New York      -0.895493
Vermont        0.500000
Florida       -0.008667
Oregon        -0.170282
Nevada        -1.000000
California     0.960208
Idaho         -1.000000
dtype: float64
```

Example: Random Sampling and Permutation

- suits = ['H', 'S', 'C', 'D']
- deck[:13]
- base_names = ['A'] + list(range(2, 11))+ ['J', 'K', 'Q']
- cards = []
- for suit in ['H', 'S', 'C', 'D']:
  - cards.extend(str(num) + suit for num in base_names)
- deck = Series(card_val, index=cards)
- deck[:13]

```
AH      1
2H      2
3H      3
4H      4
5H      5
6H      6
7H      7
8H      8
9H      9
10H    10
JH     10
KH     10
QH     10
dtype: int64
```

Example: Random Sampling and Permutation

- def draw(deck, n=5):
  - return deck.sample(n)
- draw(deck)
- get_suit = lambda card: card[-1] # last letter is suit
- c_grouped = deck.groupby(get_suit)
- for name, group in c_grouped:
  - print(name)
  - print(group)
- c_grouped.apply(draw, n=2)

```
QD      10
10C     10
AS       1
4S       4
6S       6
dtype: int64
```

```
C   KC     10
    3C      3
D   QD     10
    4D      4
H   5H      5
    AH      1
S   3S      3
    AS      1
dtype: int64
```

```
C
AC       1
2C       2
3C       3
4C       4
5C       5
6C       6
7C       7
8C       8
9C       9
10C     10
JC      10
KC      10
QC      10
dtype: int64
D
AD       1
2D       2
3D       3
4D       4
5D       5
6D       6
7D       7
8D       8
9D       9
10D     10
JD      10
KD      10
QD      10
dtype: int64
```

Example: Group Weighted Average and Correlation

- df = pd.DataFrame({'category': ['a', 'a', 'a', 'a', 'b', 'b', 'b', 'b'], 'data': np.random.randn(8), 'weights': np.random.rand(8)}); df
- get_wavg = lambda g: np.average(g['data'], weights=g['weights'])
- df.groupby('category').apply(get_wavg)

```
  category      data   weights
0        a  0.607226  0.989415
1        a  2.124653  0.245650
2        a -1.174857  0.505263
3        a -0.223118  0.844051
4        b  1.029432  0.131701
5        b -0.508956  0.564899
6        b  0.365662  0.186840
7        b  1.172103  0.967330
```

```
category
a    0.131864
b    0.567439
dtype: float64
```

Example: Group Weighted Average and Correlation

- close_px = pd.read_csv('stock_px_2.csv', parse_dates=True, index_col=0)
- close_px.info()
- close_px[-4:]

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 to 2011-10-14
Data columns (total 4 columns):
AAPL    2214 non-null float64
MSFT    2214 non-null float64
XOM     2214 non-null float64
SPX     2214 non-null float64
dtypes: float64(4)
memory usage: 86.5 KB
```

	AAPL	MSFT	XOM	SPX
2011-10-11	400.29	27.00	76.27	1195.54
2011-10-12	402.19	26.96	77.16	1207.25
2011-10-13	408.43	27.18	76.37	1203.66
2011-10-14	422.00	27.27	78.11	1224.58

Example: Group Weighted Average and Correlation

- spx_corr = lambda x: x.corrwith(x['SPX'])
- rets = close_px.pct_change().dropna()
- get_year = lambda x: x.year
- by_year = rets.groupby(get_year)
- by_year.apply(spx_corr)
- by_year.apply(lambda g: g['AAPL'].corr(g['MSFT']))

	AAPL	MSFT	XOM	SPX
2003	0.541124	0.745174	0.661265	1.0
2004	0.374283	0.588531	0.557742	1.0
2005	0.467540	0.562374	0.631010	1.0
2006	0.428267	0.406126	0.518514	1.0
2007	0.508118	0.658770	0.786264	1.0
2008	0.681434	0.804626	0.828303	1.0
2009	0.707103	0.654902	0.797921	1.0
2010	0.710105	0.730118	0.839057	1.0
2011	0.691931	0.800996	0.859975	1.0

```
2003    0.480868
2004    0.259024
2005    0.300093
2006    0.161735
2007    0.417738
2008    0.611901
2009    0.432738
2010    0.571946
2011    0.581987
dtype: float64
```

Example: Group-wise Linear Regression

- import statsmodels.api as sm
- def regress(data, yvar, xvars):
  - Y = data[yvar]
  - X = data[xvars]
  - X['intercept'] = 1.
  - result = sm.OLS(Y, X).fit()
  - return result.params
- by_year.apply(regress, 'AAPL', ['SPX'])

	SPX	intercept
2003	1.195406	0.000710
2004	1.363463	0.004201
2005	1.766415	0.003246
2006	1.645496	0.000080
2007	1.198761	0.003438
2008	0.968016	-0.001110
2009	0.879103	0.002954
2010	1.052608	0.001261
2011	0.806605	0.001514

PIVOT TABLES AND CROSS-TABULATION

Pivot Tables and Cross-Tabulation

- A pivot table is a data summarization tool frequently found in spreadsheet programs and other data analysis software
- tips.pivot_table(index=['day', 'smoker'])
- tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'], columns='smoker')

		size	tip	tip_pct	total_bill
day	smoker				
Fri	No	2.250000	2.812500	0.151650	18.420000
	Yes	2.066667	2.714000	0.174783	16.813333
Sat	No	2.555556	3.102889	0.158048	19.661778
	Yes	2.476190	2.875476	0.147906	21.276667
Sun	No	2.929825	3.167895	0.160113	20.506667
	Yes	2.578947	3.516842	0.187250	24.120000
Thur	No	2.488889	2.673778	0.160298	17.113111
	Yes	2.352941	3.030000	0.163863	19.190588

		size		tip_pct	
smoker		No	Yes	No	Yes
time	day				
Dinner	Fri	2.000000	2.222222	0.139622	0.165347
	Sat	2.555556	2.476190	0.158048	0.147906
	Sun	2.929825	2.578947	0.160113	0.187250
	Thur	2.000000	NaN	0.159744	NaN
Lunch	Fri	3.000000	1.833333	0.187735	0.188937
	Thur	2.500000	2.352941	0.160311	0.163863

Pivot Tables and Cross-Tabulation

- tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'], columns='smoker', margins=True)
- tips.pivot_table('tip_pct', index=['time', 'smoker'], columns='day', aggfunc=len, margins=True)
- tips.pivot_table('tip_pct', index=['time', 'size', 'smoker'], columns='day', aggfunc='mean', fill_value=0)

		size			tip_pct		
smoker		No	Yes	All	No	Yes	All
time	day						
Dinner	Fri	2.000000	2.222222	2.166667	0.139622	0.165347	0.158916
	Sat	2.555556	2.476190	2.517241	0.158048	0.147906	0.153152
	Sun	2.929825	2.578947	2.842105	0.160113	0.187250	0.166897
	Thur	2.000000	NaN	2.000000	0.159744	NaN	0.159744
Lunch	Fri	3.000000	1.833333	2.000000	0.187735	0.188937	0.188765
	Thur	2.500000	2.352941	2.459016	0.160311	0.163863	0.161301
All		2.668874	2.408602	2.569672	0.159328	0.163196	0.160803

day		Fri	Sat	Sun	Thur	All
time	smoker					
Dinner	No	3.0	45.0	57.0	1.0	106.0
	Yes	9.0	42.0	19.0	NaN	70.0
Lunch	No	1.0	NaN	NaN	44.0	45.0
	Yes	6.0	NaN	NaN	17.0	23.0
All		19.0	87.0	76.0	62.0	244.0

Pivot Tables and Cross-Tabulation

day			Fri	Sat	Sun	Thur
time	size	smoker				
Dinner	1	No	0.000000	0.137931	0.000000	0.000000
		Yes	0.000000	0.325733	0.000000	0.000000
	2	No	0.139622	0.162705	0.168859	0.159744
		Yes	0.171297	0.148668	0.207893	0.000000
	3	No	0.000000	0.154661	0.152663	0.000000
		Yes	0.000000	0.144995	0.152660	0.000000
	4	No	0.000000	0.150096	0.148143	0.000000
		Yes	0.117750	0.124515	0.193370	0.000000
	5	No	0.000000	0.000000	0.206928	0.000000
		Yes	0.000000	0.106572	0.065660	0.000000
	6	No	0.000000	0.000000	0.103799	0.000000
Lunch	1	No	0.000000	0.000000	0.000000	0.181728
		Yes	0.223776	0.000000	0.000000	0.000000
	2	No	0.000000	0.000000	0.000000	0.166005
		Yes	0.181969	0.000000	0.000000	0.158843
	3	No	0.187735	0.000000	0.000000	0.084246
		Yes	0.000000	0.000000	0.000000	0.204952
	4	No	0.000000	0.000000	0.000000	0.138919
		Yes	0.000000	0.000000	0.000000	0.155410
	5	No	0.000000	0.000000	0.000000	0.121389
	6	No	0.000000	0.000000	0.000000	0.173706

Cross-Tabulations: Crosstab

- A cross-tabulation (or crosstab) is a special case of a pivot table that computes group frequencies
- pd.crosstab([tips.time, tips.smoker], tips.day, margins=True)

day		Fri	Sat	Sun	Thur	All
time	smoker					
Dinner	No	3	45	57	1	106
	Yes	9	42	19	0	70
Lunch	No	1	0	0	44	45
	Yes	6	0	0	17	23
All		19	87	76	62	244

THANK YOU!!!