Data Analysis in Python Documentation

Release 0.1

Stanislav Khrapov

March 04, 2017

Contents

CHAPTER 1

Introduction

Resources

These notes are a compilation of the following original resources for the purposes of the class I teach:

- Python 3 documentation
- Python Scientific Lecture Notes
- Dive into Python 3
- Python for Econometrics
- High Performance Scientific Computing
- DataJoy documentation
- Learn Python The Hard Way
- An introduction to Numpy and Scipy

Why Python?

- 10 Reasons Python Rocks for Research (And a Few Reasons it Doesn't)
- Choosing R or Python for data analysis? An infographic
- Data Science Programming: Python vs R
- Python vs Matlab
- Infographic: Quick Guide on SAS vs R vs Python

How to start using Python

Online code editors which allow to bypass any difficulties connected with installation of Python on your own machine:

- Wakari.io (seems to be at full capacity very frequently)

For local installation it is recommended to download Anaconda distribution. Anaconda includes my favorite IDE, Spyder.

Essential libraries

- Pandas - data analysis library
- Numpy - fundamental package for scientific computing
- SciPy - numerical routines
- StatsModels - econometrics tools
- Matplotlib - plotting library
- Seaborn - pretty plotting and basic visual analysis
- Bokeh - Interactive plotting
- Plotly - Interactive plotting

Data sources

- Quandl
- Google Finance
- Yahoo Finance
- FRED

CHAPTER 2

Python basics

Contents

Your first program

Launch your favorite environment:

- **IPython** shell by typing "ipython" from a Linux/Mac terminal, or from the Windows cmd shell
- **Python** shell by typing "python" from a Linux/Mac terminal, or from the Windows cmd shell
- **Spyder** includes both **IPython** and **Python** as interactive shells

- Wakari.io has a variety of shells, including **Ipython** and **Python**

Once you have started the interpreter (wait for >>> is you use pure Python, or `In [1]:` if you use IPython), type:

```
>>> print('Hello world!')
Hello world!
```

Let's play around and see what we can get without any knowledge of programming. Try some simple math calculations:

```
>>> 2*2
4
>>> 2+3
5
>>> 4*(2+3)
20
>>> 1/2
0.5
>>> 1/3
0.3333333333333333
>>> 5-10
-5
>>> 0/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Everything seems to work as expected. The last line produces some internal complaints - we will get back to it later.

We can define some variables and ask the shell who they are:

```
>>> a = 2
>>> type(a)
<class 'int'>
>>> b = .1
>>> type(b)
<class 'float'>
>>> c = 'Hello'
>>> type(c)
<class 'str'>
```

Just to test the sanity of the language we can try adding up different variable types:

```
>>> a + b
2.1
>>> c + c
'HelloHello'
>>> a + c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Clearly, Python does not know how to add up integers to strings. Neither do we...

What if we call something that does not exist yet?

```
>>> d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'd' is not defined
```

Variable names can only contain numbers, letters (both upper and lower case), and underscores (_). They must begin with a letter or an underscore and are CaSe SeNsItIve. Some words are reserved in Python and so cannot be used for variable names.

Native Data Types

Numbers

There are three numeric types:

- integers (`int`)
- floating point (`float`)
- complex (`complex`)

Hopefully, we will not need the last one. But if you see something like `3+5j` or `6-7J`, you know you are looking at `complex` type.

Note that if you want to define a `float`, you have to use the dot (`.`), otherwise the output is an integer. For example,

```
>>> type(1)
<class 'int'>
>>> type(1.)
<class 'float'>
>>> type(float(1))
<class 'float'>
>>> type(int(1.))
<class 'int'>
>>> type(0)
<class 'int'>
>>> type(0.)
<class 'float'>
>>> type(.0)
<class 'float'>
>>> type(0.0)
<class 'float'>
```

This was extremely important in Python 2 and was the source of many inadvertent errors (try dividing 1 by 2 - you'd be surprised). With Python 3 not anymore, but the general advice of being explicit in what you mean is still there.

Division (`/`) always returns a `float`. To do floor division and get an integer result (discarding any fractional result) you can use the `//` operator; to calculate the remainder you can use `%`:

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

Notice one way of commenting your code: just use `#` after the code and before any text.

Calculating powers is done with `**` operator.

```
>>> 2**2
4
>>> 3**3
27
>>> 4**.5
2.0
```

Booleans

`bool` type is essential for any programming logic. Normally, truth and falcity are defined as `True` and `False`:

```
>>> x = True
>>> print(x)
True
>>> type(x)
<class 'bool'>
>>> y = False
>>> print(y)
False
>>> type(y)
<class 'bool'>
```

Additionally, all non-empty and non-zero values are interpreted by `bool()` function as `True`, while all empty and zero values are `False`:

```
>>> print(bool(1), bool(1.), bool(-.1))
True True True
>>> print(bool(0), bool(.0), bool(None), bool(''), bool([]))
False False False False False
```

Strings

Strings can be difined using both single (`’...’`) or double quotes (`"..."`). Backslash can be used to escape quotes.

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

The `print()` function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> print('"Isn\'t," she said.')
"Isn't," she said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
```

```
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```

If you don't want characters prefaced by \ to be interpreted as special characters, you can use *raw strings* by adding an `r` before the first quote:

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

Python is very sensitive to code aesthetics (see Style Guide). In particular, you shoud restrict yourself to 79 characters in one line! Use parenthesis to break long strings:

```
>>> text = ('Put several strings within parentheses '
            'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

Strings can be constructed using math operators and by converting numbers into strings via `str()` function:

```
>>> 2 * 'a' + '_' + 3 * 'b' + '_' + 4 * (str(.5) + '_')
'aa_bbb_0.5_0.5_0.5_0.5_'
```

Note that Python can not convert numbers into strings automatically. Unless you use `print()` function or convert explicitly.:

```
>>> 'a' + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> 'a' + str(1)
'a1'
>>> print('a', 1)
a 1
```

Lists

Lists are very convenient and simplest data containers. Here is how we store a collection of numbers in a variable:

```
>>> a = [1, 3, 5]
>>> a
[1, 3, 5]
>>> type(a)
<class 'list'>
```

Lists are not restricted to be uniform in types of their elements:

```
>>> b = [5, 2.3, 'abc', [4, 'b'], a, print]
>>> b
[5, 2.3, 'abc', [4, 'b'], [1, 3, 5], <built-in function print>]
```

Lists can be modified:

```
>>> a[1] = 4
>>> a
[1, 4, 5]
```

Lists can be merged or repeated:

```
>>> a + a
[1, 4, 5, 1, 4, 5]
>>> 3 * a
[1, 4, 5, 1, 4, 5, 1, 4, 5]
```

You can add one item to the end of the list inplace:

```
>>> a.append(7)
>>> a
[1, 4, 5, 7]
```

or add a few items:

```
>>> a.extend([0, 2])
>>> a
[1, 4, 5, 7, 0, 2]
```

Note the difference:

```
>>> a = [1, 3, 5]
>>> b = [1, 3, 5]
>>> a.append([2, 4, 6])
>>> b.extend([2, 4, 6])
>>> a
[1, 3, 5, [2, 4, 6]]
>>> b
[1, 3, 5, 2, 4, 6]
```

If the end of the list is not what you want, insert the element after a specified position:

```
>>> a.insert(1, .5)
>>> a
[1, 0.5, 4, 5, 7, 0, 2]
```

There are at least two methods to remove elements from a list:

```
>>> x = ['a', 'b', 'c', 'b']
>>> x.remove('b')
>>> x
['a', 'c', 'b']
>>> x.remove('b')
>>> x
['a', 'c']
>>> x.remove('b')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

or:

```
>>> y = ['a', 'b', 'c', 'b']
>>> y.pop()
'b'
>>> y
['a', 'b', 'c']
```

```
>>> y.pop(1)
'b'
>>> y
['a', 'c']
```

Here is how you sort a list without altering the original object, and inplace:

```
>>> x = ['a', 'b', 'c', 'b', 'a']
>>> sorted(x)
['a', 'a', 'b', 'b', 'c']
>>> x
['a', 'b', 'c', 'b', 'a']
>>> x.sort()
>>> x
['a', 'a', 'b', 'b', 'c']
```

Tuples

On the first glance tuples are very similar to lists. The difference in definition is the usage of parentheses `()` (or even without them) instead of square brackets `[]`:

```
>>> t = 12345, 54321, 'hello!'
>>> t
(12345, 54321, 'hello!')
>>> type(t)
<class 'tuple'>
>>> t = (12345, 54321, 'hello!')
>>> t
(12345, 54321, 'hello!')
```

The main difference is that tuples are *immutable* (impossible to modify):

```
>>> t[0] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Here are the reasons you want to use tuples:

- Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of a list.
- It makes your code safer if you "write-protect" data that doesn't need to be changed.
- Some tuples can be used as dictionary keys (specifically, tuples that contain immutable values like strings, numbers, and other tuples). Lists can never be used as dictionary keys, because lists are not immutable.

Dictionaries

A dictionary is an unordered set of key-value pairs. There are some restrictions on what can be a key. In general, keys can not be mutable objects. Keys must be unique. Below are a few example of dictionary initialization:

```
>>> empty_dict = dict()
>>> empty_dict
{}
>>> empty_dict = {}
>>> empty_dict
```

```
{}
>>> type(empty_dict)
<class 'dict'>
>>> grades = {'Ivan': 4, 'Olga': 5}
>>> grades
{'Ivan': 4, 'Olga': 5}
>>> grades['Petr'] = 'F'
>>> grades
{'Ivan': 4, 'Petr': 'F', 'Olga': 5}
>>> grades['Olga']
5
```

Keys and values can be accessed separately if needed:

```
>>> grades.keys()
dict_keys(['Ivan', 'Olga'])
>>> grades.values()
dict_values([4, 5])
```

Sets

A set is an unordered collection of unique values. A single set can contain values of any immutable datatype. Once you have two sets, you can do standard set operations like union, intersection, and set difference. Here is a brief demonstration:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> basket
{'orange', 'banana', 'pear', 'apple'}
>>> type(basket)
<class 'set'>
>>> 'orange' in basket
True
>>> 'crabgrass' in basket
False
```

Let’s create a second set and see what we can do with both:

```
>>> bag = {'banana', 'peach'}
>>> basket - bag
{'apple', 'orange', 'pear'}
>>> basket | bag
{'peach', 'orange', 'pear', 'banana', 'apple'}
>>> basket & bag
{'banana'}
>>> basket ^ bag
{'peach', 'apple', 'orange', 'pear'}
```

Indexing

Python data containers (including strings and lists) can be *sliced* to access their specific parts. Counting in Python starts from zero. Keep this in mind when you want to access a specific character of a string:

```
>>> word = 'Python'
>>> word[0]   # character in position 0
'P'
```

```
>>> word[5]  # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-to-last character
'o'
>>> word[-6]
'P'
```

Going beyond a single charcter:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.:

```
>>> word[:2]  # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]  # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

One could be interested only in even/odd characters in the string. In that case, we need a third index in the slice:

```
>>> word[::2]
'Pto'
>>> word[1::2]
'yhn'
```

Negative index in the third position of the slice reverses the count:

```
>>> word[::-1]
'nohtyP'
>>> word[::-2]
'nhy'
```

One way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of n characters has index n, for example:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Indexing with lists works in the same way. But on top of that, if your list contains other lists, or strings (or other **iterables**), then indexing becomes “layered”:

```
>>> x = [[1, 3, 5], ['c', 'a', 'b']]
>>> x[0][1]
3
>>> x[1][-2:]
['a', 'b']
```

Control flow

if/elif/else

Writing conditional statements in Python is very easy. Start from `if`, continue with `elif`, and finish with `else`. For example,

```
In [1]: if 2**2 == 4:
   ...:     print('Should be True')
   ...:
Should be True
```

Be careful to **respect the indentation depth**. The Ipython shell automatically increases the indentation depth after a column **:** sign; to decrease the indentation depth, go four spaces to the left with the Backspace key. Press the Enter key twice to leave the logical block.

```
In [1]: a = 10

In [2]: if a == 1:
   ...:     print(1)
   ...: elif a == 2:
   ...:     print(2)
   ...: elif a == 3:
   ...:     print(3)
   ...: else:
   ...:     print('A lot')
   ...:
A lot
```

Besides checking for equality as in the previous examples, you can check for other statements evaluating to `bool`. These are comparison operators: `<, >, <=, =>`. Testing for equality of two objects is done with `is` operator:

```
>>> a, b = 1, 1.
>>> a == b
True
>>> a is b
False
```

You can test whether an object belongs to a collection using `in` operator. Note that if a collection is of type `dict`, then the search is done over dictionaries:

```
>>> a = [1, 2, 4]
>>> 2 in a, 4 in a
(True, True)
>>> b = {'a': 3, 'c': 8}
>>> 'c' in b
True
```

Loops

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```
In [7]: for i in range(4):
   ...:     print(i)
   ...:
0
```

```
1
2
3
```

The for statement in Python differs a bit from what you may be used to in other programming languages. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as in C), Python's for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.

```
In [6]: words = ['cat', 'window', 'bird']
   ...: for w in words:
   ...:     print(w, len(w))
   ...:
cat 3
window 6
bird 4
```

Here is another example.

```
In [1]: for letter in 'Python':
   ...:     print(letter)
   ...:
P
y
t
h
o
n
```

Coming back to range() function. It can have at most three arguments, range(first, last, step). Given this knowledge we can generate various sequences. Note that this function returns neither a list not a tuple. In fact, it is an object itself. In order to check what are the indices if we use range in a for loop, we can convert it to list using list() function. The reason behind this behavior is to save memory: range does not store the whole list, only its definition.

```
>>> list(range(2, 10))
[2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2, 10, 3))
[2, 5, 8]
>>> list(range(-2, -10, -3))
[-2, -5, -8]
```

If you need to break out of the loop or skip an iteration, then you need to know two statements, break and continue, respectively.

```
In [3]: a = [1, 0, 2, 4]
   ...: for element in a:
   ...:     if element == 0:
   ...:         continue
   ...:     print(1. / element)
   ...:
1.0
0.5
0.25
```

or

```
In [4]: a = [1, 0, 2, 4]
   ...: for element in a:
```

```
   ...:     if element == 0:
   ...:         break
   ...:     print(1. / element)
   ...:
1.0
```

Common use case is to iterate over items while keeping track of current index. Quick and dirty way to do this is:

```
In [5]: words = ('cool', 'powerful', 'readable')
   ...: for i in range(0, len(words)):
   ...:     print(i, words[i])
   ...:
0 cool
1 powerful
2 readable
```

Yet, Python provides a much more elegant approach:

```
In [7]: for index, item in enumerate(words):
   ...:     print(index, item)
   ...:
0 cool
1 powerful
2 readable
```

Try iterating over dictionaries yourslef. You should find out that Python iterates over keys only. In order to have access to the whole pair, one should use `items()` method:

```
In [1]: grades = {'Ivan': 4, 'Olga': 5, 'Petr': 4.5}
   ...: for key, val in grades.items():
   ...:     print('%s has grade: %s' % (key, val))
   ...:
Ivan has grade: 4
Petr has grade: 4.5
Olga has grade: 5
```

Here is how you might compute Pi:

```
In [1]: pi = 2
   ...: for i in range(1, 1000):
   ...:     pi *= 4*i**2 / (4*i**2 - 1)
   ...: print(pi)
   ...:
3.1408069608284657
```

Or if you want to stop after certain precision was achieved (a common use case), you might want to use `while` loop:

```
In [3]: pi, error, i = 2, 1e10, 1
   ...: while error > 1e-3:
   ...:     pi *= 4*i**2 / (4*i**2 - 1)
   ...:     error = abs(pi - 3.141592653589793)
   ...:     i += 1
   ...: print(pi)
   ...:
3.1405927760475945
```

List comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
In [1]: squares = []
   ...: for x in range(10):
   ...:     squares.append(x**2)
   ...: print(squares)
   ...:
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

As always, Python has a more elegant solution with the same result:

```
>>> squares = [x**2 for x in range(10)]
```

List comprehensions can include more `for` statements and even `if` statements:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

which creates a list of pairs with distinct elements. Equivalenly, one could write this over several lines:

```
In [1]: combs = []
   ...: for x in [1,2,3]:
   ...:     for y in [3,1,4]:
   ...:         if x != y:
   ...:             combs.append((x, y))
   ...: print(combs)
```

Below are a few more examples:

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Finally, we can transpose a “matrix” represented as a list of lists in the following several ways.

```
In [1]: matrix = [
   ...:     [1, 2, 3, 4],
   ...:     [5, 6, 7, 8],
   ...:     [9, 10, 11, 12],
   ...: ]
```

First, the longest but clearest:

```
In [1]: transposed = []
   ...: for i in range(4):
   ...:     transposed_row = []
   ...:     for row in matrix:
   ...:         transposed_row.append(row[i])
   ...:     transposed.append(transposed_row)
   ...: print(transposed)
   ...:
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Next uses one list comprehension:

```
In [1]: transposed = []
   ...: for i in range(4):
   ...:     transposed.append([row[i] for row in matrix])
```

Or, one single nested list comprehension:

```
>>> [[row[i] for row in matrix] for i in range(4)]
```

And, finally, the most elegant (in the context of standard library) way:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Functions

Functions are well defined logically complete blocks of actions combined to serve a specific purpose. Functions are separated from the main script to be reused again and again in other projects.

Function definition

The simplest function definition is illustrated in the following example:

```
def simplest_function():
    print('I\'m your function!')
```

which after calling produces the following output:

```
>>> simplest_function()
I'm your function!
```

The keyword `def` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

A slightly more complicated example to compute Fibbonaci series:

```
def fib(n):
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
```

Let’s try and call this function to find out all Fibonacci numbers up to 2000:

```
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Notice the line below function name in tripled quotes. This is called *docstring*. We will come back to it in *Documenenting your code*.

The result of running function above is just a screen output. If we try to assign the result of this function to a new variable, we will only get `None`:

```
>>> out = fib(0)
>>> print(out)
None
```

What if you want to store the result? Then you have to use `return` statement and say explicitely what your function should produce in the end.

```
def fib(n):
    """Print a Fibonacci series up to n and return the result."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Now let’s try this function instead:

```
>>> out = fib(2000)
>>> print(out)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597]
```

Now variable `out` is non-empty. It holds the `list` of Fibonacci numbers.

Above examples have shown how to define functions without any arguments and with just one argument. In fact, function definition is much more flexible than that. Read on.

Positional arguments

Passing several arguments to a function is done with their order in mind.

```
def power(x, a):
    """Take a power"""
    return x**a
```

If you make a mistake in the order of arguments, the function has no way to see that:

```
>>> print(power(2, 3), power(3, 2))
8 9
```

Default argument values

Some arguments may have default values. This is used to simplify function calls especially if arguments are numerous. Default arguments always follow positional ones.

```
def power(x, a=2):
    """Take a power"""
    return x**a
```

Here is how you call it:

```
>>> print(power(2), power(3))
4 9
```

The default values are evaluated once at function definition.

```
i = 5

def fun(arg=i):
    print(arg)

i = 6
```

The call to this function prduces:

```
>>> fun()
5
```

The side effect is that the default value is shared between the calls:

```
def fun(a, L=[]):
    L.append(a)
    return L

print(fun(1))
print(fun(2))
print(fun(3))
```

This prints

```
[1]
[1, 2]
[1, 2, 3]
```

Here is one possible way to overcome this:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Keyword arguments

If keeping the order of the arguments becomes a problem, then keyword (or optional) arguments are here to help. These are the same arguments with default values but redefined in function calls.

```
def slicer(seq, start=None, stop=None, step=None):
        return seq[start:stop:step]
```

This function has three default values. They all follow the variable without default. Here are a few examples of using this function:

```
>>> print(rhyme)
['one', 'fish,', 'two', 'fish,', 'red', 'fish,', 'blue', 'fish']
>>> print(slicer(rhyme))
['one', 'fish,', 'two', 'fish,', 'red', 'fish,', 'blue', 'fish']
>>> print(slicer(rhyme, step=2))
['one', 'two', 'red', 'blue']
>>> print(slicer(rhyme, 1, step=2))
['fish,', 'fish,', 'fish,', 'fish']
>>> print(slicer(rhyme, stop=4, step=2, start=1))
['fish,', 'fish,']
>>> print(slicer(rhyme, 1, 4, 2))
['fish,', 'fish,']
```

The following are invalid calls:

```
>>> slicer()                      # required argument missing
>>> slicer(start=2, 'Python')     # non-keyword argument after a keyword argument
>>> slicer('Python', 2, start=3) # duplicate value for the same argument
>>> slicer(actor='John Cleese')   # unknown keyword argument
```

Arbitrary argument lists

If you do not know in advance how many arguments you will need to pass to a function, then you can use function definition as follows:

```
def fun(var, *args, **kwargs):
    print('First mandatory argument:', var)
    if len(args) > 0:
        print('\nOptional positional arguments:')
    for idx, arg in enumerate(args):
        print('Argument number "%s" is "%s"' % (idx, arg))
    if len(kwargs) > 0:
        print('\nOptional keyword arguments:')
    for key, value in kwargs.items():
        print('Argument called "%s" is "%s"' % (key, value))
```

Calling this function produces:

```
>>> fun(2, 'a', 'Python', method='OLS', limit=1e2)
First mandatory argument: 2

Optional positional arguments:
Argument number "0" is "a"
Argument number "1" is "Python"

Optional keyword arguments:
Argument called "method" is "OLS"
Argument called "limit" is "100.0"
```

At the same time, calling this function with the only mandatory argument results in a much simple output:

```
>>> fun(2)
First mandatory argument: 2
```

Placing a star in front of `args` makes interpreter to expect a tuple of arbitrary length which is then unpacked to separate arguments. Placing two stars in front of `kwargs` makes Python unpack it as a dictionary into key-value pairs. So, you can pass arguments as tuples and dictionaries which sometimes significantly improves readability of the code. The following lines produce the same output as in the first example of this subsection:

```
>>> args = ('a', 'Python')
>>> kwargs = {'method': 'OLS', 'limit': 1e2}
>>> fun(2, *args, **kwargs)
```

Lambda functions

Small anonymous functions can be created with the `lambda` keyword. Lambda functions can be used wherever function objects are required. They are restricted to be one-liners. Here is an example of a function that returns another function:

```
def make_power(n):
    return lambda x: x ** n
```

And the way to use it is as follows:

```
>>> power = make_power(3)
>>> print(power(0), power(2))
0 8
```

Another example shows how to pass a function as an argument without formally defining it:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Passing by value

In Python parameters to functions are references to objects, which are passed by value. When you pass a variable to a function, Python passes the reference to the object to which the variable refers (the value). Not the variable itself.

If the value passed in a function is immutable, the function does not modify the caller's variable. If the value is mutable, the function may modify the caller's variable in-place:

```
def try_to_modify(x, y, z):
    x = 23 # immutable object
    y.append(42)
    z = [99] # reference to new object
    print(x, y, z)
```

Here is what happens if we call this function:

```
>>> a = 77     # immutable variable
>>> b = [99]   # mutable variable
>>> c = [28]
>>> try_to_modify(a, b, c)
23 [99, 42] [99]
>>> print(a, b, c)
77 [99, 42] [28]
```

Classes

Python supports object-oriented programming (OOP). The goals of OOP are:

- to organize the code, and
- to re-use code in similar contexts.

Click here for further details.

Modules and Packages

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents:

```
# Fibonacci numbers module

def fib(n):
    """write Fibonacci series up to n"""
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n):
    """return Fibonacci series up to n"""
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not enter the names of the functions defined in `fibo` directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement.

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

There is a variant of the `import` statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

Probably the safest way of importing objects from modules is through a short reference to the module name:

```
>>> import fibo as fb
>>> fb.fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This way you may safely define another `fibo` function and it will not create any conflict with unambiguously different function `fb.fib`.

When you run a Python module with:

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to "__main__". That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the "main" file:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

For more details click here.

Documenenting your code

A Guide to NumPy/SciPy Documentation

CHAPTER 3

NumPy. Manipulations with numerical data

Contents

Array creation

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. Arrays make operations with large amounts of numeric data very fast and are generally much more efficient than lists.

It is a convention to import NumPy as follows:

```
>>> import numpy as np
```

The simplest way to create an elementary array is from a list:

```
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
>>> type(a)
<type 'numpy.ndarray'>
>>> a.dtype
dtype('int64')
```

The type of the array can also be explicitly specified at creation time:

```
>>> a = np.array([0, 1, 2, 3], float)
>>> a.dtype
dtype('float64')
```

Array transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

```
>>> b = np.array([[1.5,2,3], [4,5,6]])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
>>> c = np.array([[[1], [2]], [[3], [4]]])
>>> c
array([[[1],
        [2]],

       [[3],
        [4]]])
>>> print(a.ndim, b.ndim, c.ndim)
1 2 3
>>> print(a.shape, b.shape, c.shape)
(4,) (2, 3) (2, 2, 1)
```

There are a lot of functions to create some standard arrays, such as filled with ones, zeros, etc.

```
>>> a = np.arange(10)
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(1, 9, 2) # start, end (exclusive), step
array([1, 3, 5, 7])
>>> np.linspace(0, 1, 6) # start, end, num-points
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
>>> np.linspace(0, 1, 5, endpoint=False)
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
>>> np.ones((3, 3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.zeros((2, 2))
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> np.diag(np.array([1, 2, 3, 4]))
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
>>> np.ones_like(np.zeros((2, 2)))
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> np.zeros_like(np.ones((2, 2)))
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> np.random.rand(4) # U[0, 1]
array([ 0.37534773,  0.19079141,  0.80011337,  0.54003586])
>>> np.random.randn(4) # N(0, 1)
```

```
array([-0.2981319 , -0.06627354,  0.31080455,  0.28470444])
```

One-dimensional versions of multi-dimensional arrays can be generated with flatten:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> a.flatten()
array([ 1.,  2.,  3.,  4.,  5.,  6.])
```

Two or more arrays can be concatenated together using the concatenate function with a tuple of the arrays to be joined:

```
>>> a = np.array([1, 2], float)
>>> b = np.array([3, 4, 5, 6], float)
>>> c = np.array([7, 8, 9], float)
>>> np.concatenate((a, b, c))
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

If an array has more than one dimension, it is possible to specify the axis along which multiple arrays are concatenated. By default (without specifying the axis), NumPy concatenates along the first dimension:

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> b = np.array([[5, 6], [7, 8]], float)
>>> np.concatenate((a,b))
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.],
       [ 7.,  8.]])
>>> np.concatenate((a, b), axis=0)
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.],
       [ 7.,  8.]])
>>> np.concatenate((a, b), axis=1)
array([[ 1.,  2.,  5.,  6.],
       [ 3.,  4.,  7.,  8.]])
```

Finally, the dimensionality of an array can be increased using the newaxis constant in bracket notation:

```
>>> a = np.array([1, 2, 3], float)
>>> a
array([1., 2., 3.])
>>> a[:,np.newaxis]
array([[ 1.],
       [ 2.],
       [ 3.]])
>>> a[:, np.newaxis].shape
(3, 1)
>>> b[np.newaxis, :]
array([[ 1.,  2.,  3.]])
>>> b[np.newaxis, :].shape
(1, 3)
```

Indexing, Slicing

The items of an array can be accessed and assigned to the same way as other Python sequences (e.g. lists):

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[0], a[2], a[-1]
(0, 2, 9)
```

Similarly, array order can be reversed:

```
>>> a[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

For multidimensional arrays:

```
>>> a = np.diag(np.arange(3))
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 2]])
>>> a[1, 1]
1
>>> a[2, 1] = 10
>>> a
array([[ 0,  0,  0],
       [ 0,  1,  0],
       [ 0, 10,  2]])
>>> a[1]
array([0, 1, 0])
```

Arrays, like other Python sequences can also be sliced:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # [start:end:step]
array([2, 5, 8])
```

All three slice components are not required: by default, `start` is 0, `end` is the last and `step` is 1:

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

A more sophisticated example for multidimensional array:

```
>>> a = np.arange(60).reshape((6, 10))[:, :6]
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
>>> a[0, 3:5]
array([3, 4])
>>> a[4:, 5:]
array([[45],
       [55]])
```

```
>>> a[4:, 4:]
array([[44, 45],
       [54, 55]])
>>> a[:, 2]
array([ 2, 12, 22, 32, 42, 52])
>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

Arrrays can be sliced using boolean logic:

```
>>> np.random.seed(3)
>>> a = np.random.random_integers(0, 20, 15)
>>> a
array([10,  3,  8,  0, 19, 10, 11,  9, 10,  6,  0, 20, 12,  7, 14])
>>> (a % 3 == 0)
array([False,  True, False,  True, False, False, False,  True, False,
        True,  True, False,  True, False, False], dtype=bool)
>>> a[a % 3 == 0]
array([ 3,  0,  9,  6,  0, 12])
>>> a[a % 3 == 0] = -1
>>> a
array([10, -1,  8, -1, 19, 10, 11, -1, 10, -1, -1, 20, -1,  7, 14])
```

Copies and Views

When operating and manipulating arrays, their data is sometimes copied into a new array and sometimes not. This is often a source of confusion for beginners. There are three cases:

No Copy at All

Simple assignments make no copy of array objects or of their data.:

```
>>> a = arange(12)
>>> b = a   # no new object is created
>>> b is a  # a and b are two names for the same ndarray object
True
>>> b.shape = 3, 4   # changes the shape of a
>>> a.shape
(3, 4)
```

Python passes mutable objects as references, so function calls make no copy.

```
>>> def f(x):
...     # id is a unique identifier of an object
...     print id(x)
...
>>> id(a)
148293216
>>> f(a)
148293216
```

View or Shallow Copy

Different array objects can share the same data. The view method creates a new array object that looks at the same data.

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a  # c is a view of the data owned by a
True
>>> c.flags.owndata
False
>>> c.shape = 2, 6  # a's shape doesn't change
>>> a.shape
(3, 4)
>>> c[0, 4] = 1234  # a's data changes
>>> a
array([[   0,    1,    2,    3],
       [1234,    5,    6,    7],
       [   8,    9,   10,   11]])
```

Slicing an array returns a view of it:

```
>>> s = a[:, 1:3]
>>> s[:] = 10  # s[:] is a view of s. Note the difference between s=10 and s[:]=10
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

Deep Copy

The copy method makes a complete copy of the array and its data.:

```
>>> d = a.copy()  # a new array object with new data is created
>>> d is a
False
>>> d.base is a  # d doesn't share anything with a
False
>>> d[0, 0] = 9999
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

Array manipulations

Basic operations

With scalars:

```
>>> a = np.array([1, 2, 3, 4])
>>> a + 1
array([2, 3, 4, 5])
```

```
>>> 2**a
array([ 2,  4,  8, 16])
```

All arithmetic operates elementwise:

```
>>> b = np.ones(4) + 1
>>> a - b
array([-1.,  0.,  1.,  2.])
>>> a * b
array([ 2.,  4.,  6.,  8.])
>>> c = np.arange(5)
>>> 2**(c + 1) - c
array([ 2,  3,  6, 13, 28])
```

Array multiplication is not matrix multiplication:

```
>>> c = np.ones((3, 3))
>>> c * c
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> c.dot(c)
array([[ 3.,  3.,  3.],
       [ 3.,  3.,  3.],
       [ 3.,  3.,  3.]])
```

Other operations

Comparisons:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> a == b
array([False,  True, False,  True], dtype=bool)
>>> a > b
array([False, False,  True, False], dtype=bool)
```

Array-wise comparisons:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> c = np.array([1, 2, 3, 4])
>>> np.array_equal(a, b)
False
>>> np.array_equal(a, c)
True
>>> np.allclose(a, a + a*1e-5)
True
>>> np.allclose(a, a + a*1e-4)
False
```

Logical operations:

```
>>> a = np.array([1, 1, 0, 0], dtype=bool)
>>> b = np.array([1, 0, 1, 0], dtype=bool)
>>> np.logical_or(a, b)
array([ True,  True,  True, False], dtype=bool)
```

```
>>> np.logical_and(a, b)
array([ True, False, False, False], dtype=bool)
```

The `where` function forms a new array from two arrays of equivalent size using a Boolean filter to choose between elements of the two. Its basic syntax is `where(boolarray, truearray, falsearray)`:

```
>>> a = np.array([1, 3, 0], float)
>>> np.where(a != 0, 1 / a, a)
array([ 1. , 0.33333333, 0. ])
```

Broadcasting can also be used with the where function:

```
>>> np.where(a > 0, 3, 2)
array([3, 3, 2])
```

Broadcasting

Arrays that do not match in the number of dimensions will be broadcasted by Python to perform mathematical operations. This often means that the smaller array will be repeated as necessary to perform the operation indicated. Consider the following:

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> b = np.array([-1, 3], float)
>>> a
array([[ 1., 2.],
       [ 3., 4.],
       [ 5., 6.]])
>>> b
array([-1., 3.])
>>> a + b
array([[ 0., 5.],
       [ 2., 7.],
       [ 4., 9.]])
```

Here, the one-dimensional array `b` was broadcasted to a two-dimensional array that matched the size of `a`. In essence, `b` was repeated for each item in `a`, as if it were given by:

```
array([[-1., 3.],
       [-1., 3.],
       [-1., 3.]])
```

Python automatically broadcasts arrays in this manner. Sometimes, however, how we should broadcast is ambiguous. In these cases, we can use the newaxis constant to specify how we want to broadcast:

```
>>> a = np.zeros((2,2), float)
>>> b = np.array([-1., 3.], float)
>>> a
array([[ 0., 0.],
       [ 0., 0.]])
>>> b
array([-1., 3.])
>>> a + b
array([[-1., 3.],
       [-1., 3.]])
>>> a + b[np.newaxis, :]
array([[-1., 3.],
       [-1., 3.]])
>>> a + b[:, np.newaxis]
```

```
array([[-1., -1.],
       [ 3.,  3.]])
```

Reductions

We can easily compute sums and products:

```
>>> a = np.array([2, 4, 3])
>>> a.sum(), a.prod()
(9, 24)
>>> np.sum(a), np.prod(a)
(9, 24)
```

Some basic statistics:

```
>>> a = np.random.randn(100)
>>> a.mean()
-0.083139603089394359
>>> np.median(a)
-0.14321054235009417
>>> a.std()
1.0565446101521685
>>> a.var()
1.1162865132415978
>>> a.min(), a.max()
(-2.9157377517927121, 2.1581493420569187)
>>> np.percentile(a, [5, 50, 95])
array([-1.48965296, -0.08633928,  1.36836205])
```

For multidimensional arrays, each of the functions thus far described can take an optional argument `axis` that will perform an operation along only the specified axis, placing the results in a return array:

```
>>> a = np.array([[0, 2], [3, -1], [3, 5]], float)
>>> a.mean(axis=0)
array([ 2.,  2.])
>>> a.mean(axis=1)
array([ 1.,  1.,  4.])
>>> a.min(axis=1)
array([ 0., -1.,  3.])
>>> a.max(axis=0)
array([ 3.,  5.])
```

It is possible to find the index of the smallest and largest element:

```
>>> a = np.array([2, 1, 9], float)
>>> a.argmin()
1
>>> a.argmax()
2
```

Like lists, arrays can be sorted:

```
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> sorted(a)
[-1.0, 0.0, 2.0, 5.0, 6.0]
>>> a.sort()
>>> a
array([-1., 0., 2., 5., 6.])
```

Values in an array can be “clipped” to be within a prespecified range. This is the same as applying `min(max(x, minval), maxval)` to each element `x` in an array.

```
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> a.clip(0, 5)
array([ 5., 2., 5., 0., 0.])
```

Unique elements can be extracted from an array:

```
>>> a = np.array([1, 1, 4, 5, 5, 5, 7], float)
>>> np.unique(a)
array([ 1., 4., 5., 7.])
```

CHAPTER 4

Pandas. Data processing

Pandas is an essential data analysis library within Python ecosystem. For more details read Pandas Documentation.

Contents

Data structures

Pandas operates with three basic datastructures: *Series*, *DataFrame*, and *Panel*. There are extensions to this list, but for the purposes of this material even the first two are more than enough.

We start by importing NumPy and Pandas using their conventional short names:

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

In [2]: import pandas as pd

In [3]: randn = np.random.rand # To shorten notation in the code that follows
```

Series

Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index. The basic method to create a Series is to call:

```
>>> s = Series(data, index=index)
```

The first mandatory argument can be

- array-like
- dictionary
- scalar

Array-like

If `data` is an array-like, `index` must be the same length as `data`. If no index is passed, one will be created having values `[0, ..., len(data) - 1]`.

```
In [4]: s = pd.Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [5]: s
Out[5]:
a    0.261916
b    0.937129
c    0.418654
d    0.719897
e    0.343347
dtype: float64

In [6]: s.index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[6]

In [7]: pd.Series(randn(5))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
0    0.000740
1    0.508279
2    0.116335
3    0.832410
4    0.960953
dtype: float64
```

Dictionary

Dictionaries already have a natural candidate for the index, so passing the `index` separately seems redundant, although possible.

```
In [8]: d = {'a' : 0., 'b' : 1., 'c' : 2.}

In [9]: pd.Series(d)
Out[9]:
a    0.0
b    1.0
c    2.0
dtype: float64

In [10]: pd.Series(d, index=['b', 'c', 'd', 'a'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[10]:
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

Scalar

If `data` is a scalar value, an index must be provided. The value will be repeated to match the length of index.

```
In [11]: pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
Out[11]:
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

Series is similar to array

Slicing and other operations on *Series* produce very similar results to those on `array` but with a twist. Index is also sliced and always remain a part of a data container.

```
In [12]: s[0]
Out[12]: 0.26191552776599869

In [13]: s[:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[13]:
a    0.261916
b    0.937129
c    0.418654
dtype: float64

In [14]: s[s > s.median()]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[1
b    0.937129
d    0.719897
dtype: float64

In [15]: s[[4, 3, 1]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
e    0.343347
d    0.719897
```

```
b    0.937129
dtype: float64
```

Similarly to NumPy arrays, Series can be used to speed up loops by using vectorization.

```
In [16]: s + s
Out[16]:
a    0.523831
b    1.874258
c    0.837307
d    1.439795
e    0.686694
dtype: float64

In [17]: s * 2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[1
a    0.523831
b    1.874258
c    0.837307
d    1.439795
e    0.686694
dtype: float64

In [18]: np.exp(s)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
a    1.299417
b    2.552643
c    1.519914
d    2.054222
e    1.409658
dtype: float64
```

A key difference between Series and array is that operations between Series automatically align the data based on label. Thus, you can write computations without giving consideration to whether the Series involved have the same labels.

```
In [19]: s[1:] + s[:-1]
Out[19]:
a         NaN
b    1.874258
c    0.837307
d    1.439795
e         NaN
dtype: float64
```

The result of an operation between unaligned Series will have the union of the indexes involved. If a label is not found in one Series or the other, the result will be marked as missing NaN. Being able to write code without doing any explicit data alignment grants immense freedom and flexibility in interactive data analysis and research. The integrated data alignment features of the pandas data structures set pandas apart from the majority of related tools for working with labeled data.

Series is similar to dictionary

A few examples to illustrate the heading.

```
In [20]: s['a']
Out[20]: 0.26191552776599869
```

```
In [21]: s['e'] = 12.

In [22]: s
Out[22]:
a     0.261916
b     0.937129
c     0.418654
d     0.719897
e    12.000000
dtype: float64

In [23]: 'e' in s
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\O

In [24]: 'f' in s
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
```

Name attribute

Series can also have a name attribute which will become very useful when summarizing data with tables and plots.

```
In [25]: s = pd.Series(np.random.randn(5), name='random series')

In [26]: s
Out[26]:
0   -0.678101
1   -0.664686
2   -0.775879
3   -0.470269
4   -0.375136
Name: random series, dtype: float64

In [27]: s.name
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
```

DataFrame

DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. Like Series, DataFrame accepts many different kinds of input:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- A Series
- Another DataFrame

Along with the data, you can optionally pass **index** (row labels) and **columns** (column labels) arguments. If you pass an index and / or columns, you are guaranteeing the index and / or columns of the resulting DataFrame. Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.

If axis labels are not passed, they will be constructed from the input data based on common sense rules.

From dict of Series or dicts

The result index will be the union of the indexes of the various Series. If there are any nested dicts, these will be first converted to Series. If no columns are passed, the columns will be the sorted list of dict keys.

```
In [28]: d = {'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
   ....:      'two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
   ....:

In [29]: df = pd.DataFrame(d)

In [30]: df
Out[30]:
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0

In [31]: pd.DataFrame(d, index=['d', 'b', 'a'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[31]:
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0

In [32]: pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
```

The row and column labels can be accessed respectively by accessing the index and columns attributes:

```
In [33]: df.index
Out[33]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [34]: df.columns
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[34]: Index(['one', 'two'], dtype='object')
```

From dict of array-likes

The ndarrays must all be the same length. If an index is passed, it must clearly also be the same length as the arrays. If no index is passed, the result will be `range(n)`, where `n` is the array length.

```
In [35]: d = {'one' : [1., 2., 3., 4.], 'two' : [4., 3., 2., 1.]}

In [36]: pd.DataFrame(d)
Out[36]:
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0

In [37]: pd.DataFrame(d, index=['a', 'b', 'c', 'd'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[37]:
   one  two
a  1.0  4.0
b  2.0  3.0
c  3.0  2.0
d  4.0  1.0
```

From a list of dicts

```
In [38]: data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

In [39]: pd.DataFrame(data2)
Out[39]:
   a   b     c
0  1   2   NaN
1  5  10  20.0

In [40]: pd.DataFrame(data2, index=['first', 'second'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[40]:
        a   b     c
first   1   2   NaN
second  5  10  20.0

In [41]: pd.DataFrame(data2, columns=['a', 'b'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
   a   b
0  1   2
1  5  10
```

From a dict of tuples

```
In [42]: pd.DataFrame({('a', 'b'): {('A', 'B'): 1, ('A', 'C'): 2},
   ....:               ('a', 'a'): {('A', 'C'): 3, ('A', 'B'): 4},
   ....:               ('a', 'c'): {('A', 'B'): 5, ('A', 'C'): 6},
   ....:               ('b', 'a'): {('A', 'C'): 7, ('A', 'B'): 8},
   ....:               ('b', 'b'): {('A', 'D'): 9, ('A', 'B'): 10}})
   ....:
Out[42]:
       a              b
       a    b    c    a     b
A B  4.0  1.0  5.0  8.0  10.0
  C  3.0  2.0  6.0  7.0   NaN
  D  NaN  NaN  NaN  NaN   9.0
```

From a Series

The result will be a DataFrame with the same index as the input Series, and with one column whose name is the original name of the Series (only if no other column name provided).

Basic functionality

Here are the data sets that will be used below.

```
In [43]: index = pd.date_range('1/1/2000', periods=8)

In [44]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [45]: df = pd.DataFrame(np.random.randn(8, 3), index=index,
   ....:                   columns=['A', 'B', 'C'])
   ....:
```

Head and Tail

To view a small sample of a Series or DataFrame object, use the `head()` and `tail()` methods. The default number of elements to display is five, but you may pass a custom number.

```
In [46]: long_series = pd.Series(np.random.randn(1000))

In [47]: long_series.head()
Out[47]:
0   -0.780819
1    0.631099
2    0.416289
3    1.176433
4   -0.116158
dtype: float64

In [48]: long_series.tail(3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[48
997   -0.733624
998   -0.843908
999   -1.428264
dtype: float64
```

Attributes and the raw values

Pandas objects have a number of attributes enabling you to access the metadata

- `shape`: gives the axis dimensions of the object, consistent with ndarray
- Axis labels
  - Series: `index` (only axis)
  - DataFrame: `index` (rows) and `columns`

Note, these attributes can be safely assigned to!

```
In [49]: df[:2]
Out[49]:
                   A         B         C
2000-01-01 -0.019051  1.754216 -0.683589
2000-01-02 -0.040374 -1.482709 -0.273475

In [50]: df.columns = [x.lower() for x in df.columns]

In [51]: df
Out[51]:
                   a         b         c
2000-01-01 -0.019051  1.754216 -0.683589
```

```
2000-01-02 -0.040374 -1.482709 -0.273475
2000-01-03 -1.495000 -1.220174  0.549418
2000-01-04  0.610248 -0.086783 -1.162908
2000-01-05  0.594158 -0.237087 -0.943017
2000-01-06  0.407705  0.210901  0.434077
2000-01-07  0.550795 -0.392167 -0.857879
2000-01-08  0.305878  0.924261  0.259116
```

To get the actual data inside a data structure, one need only access the values property:

```
In [52]: s.values
Out[52]: array([ 0.09233447, -0.94042079,  0.3506382 , -0.59325573, -1.00153266])

In [53]: df.values
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[53]:
array([[-0.01905145,  1.75421565, -0.68358856],
       [-0.04037431, -1.48270934, -0.27347476],
       [-1.49500022, -1.22017434,  0.54941815],
       [ 0.61024815, -0.086783  , -1.16290799],
       [ 0.59415766, -0.23708742, -0.94301684],
       [ 0.40770453,  0.21090147,  0.43407654],
       [ 0.55079505, -0.39216667, -0.85787884],
       [ 0.30587792,  0.9242608 ,  0.25911602]])
```

Descriptive statistics

A large number of methods for computing descriptive statistics and other related operations on Series and DataFrame. Most of these are aggregations (hence producing a lower-dimensional result) like `sum()`, `mean()`, and `quantile()`, but some of them, like `cumsum()` and `cumprod()`, produce an object of the same size. Generally speaking, these methods take an axis argument, just like `ndarray.{sum, std, ...}`, but the axis can be specified by name or integer:

- Series: no axis argument needed
- DataFrame: "index" (axis=0, default), "columns" (axis=1)

```
In [54]: df = pd.DataFrame({'one' : pd.Series(np.random.randn(3), index=['a', 'b', 'c']),
   ....:                    'two' : pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd']),
   ....:                    'three' : pd.Series(np.random.randn(3), index=['b', 'c', 'd'])})
   ....:

In [55]: df.mean(0)
Out[55]:
one     -0.162649
three   -0.168463
two      0.602863
dtype: float64

In [56]: df.mean(1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[56]:
a   -0.537465
b    0.221972
c    0.230379
d    0.567997
dtype: float64
```

All such methods have a `skipna` option signaling whether to exclude missing data (`True` by default):

```
In [57]: df.sum(0, skipna=False)
Out[57]:
one             NaN
three           NaN
two        2.411452
dtype: float64

In [58]: df.sum(axis=1, skipna=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[58]:
a   -1.074930
b    0.665916
c    0.691136
d    1.135994
dtype: float64
```

Combined with the broadcasting / arithmetic behavior, one can describe various statistical procedures, like standardization (rendering data zero mean and standard deviation 1), very concisely:

```
In [59]: ts_stand = (df - df.mean()) / df.std()

In [60]: ts_stand.std()
Out[60]:
one      1.0
three    1.0
two      1.0
dtype: float64

In [61]: xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)

In [62]: xs_stand.std(1)
Out[62]:
a    1.0
b    1.0
c    1.0
d    1.0
dtype: float64
```

Series also has a method `nunique()` which will return the number of unique non-null values:

```
In [63]: series = pd.Series(np.random.randn(500))

In [64]: series[20:500] = np.nan

In [65]: series[10:20] = 5

In [66]: series.nunique()
Out[66]: 11
```

Summarizing data: `describe`

There is a convenient `describe()` function which computes a variety of summary statistics about a Series or the columns of a DataFrame:

```
In [67]: series = pd.Series(np.random.randn(1000))

In [68]: series[::2] = np.nan
In [69]: series.describe()
Out[69]:
count    500.000000
mean      -0.035852
std        1.017672
min       -3.870073
25%       -0.645819
50%       -0.051329
75%        0.646074
max        3.246603
dtype: float64

In [70]: frame = pd.DataFrame(np.random.randn(1000, 5),
   ....:                      columns=['a', 'b', 'c', 'd', 'e'])
   ....:

In [71]: frame.ix[::2] = np.nan

In [72]: frame.describe()
Out[72]:
                a           b           c           d           e
count  500.000000  500.000000  500.000000  500.000000  500.000000
mean     0.034814    0.038012   -0.022464    0.038878   -0.025334
std      0.940658    0.970266    0.974810    0.976934    1.002783
min     -2.983143   -2.839655   -2.490240   -2.700005   -2.898009
25%     -0.630357   -0.701047   -0.652933   -0.657623   -0.729624
50%      0.003269    0.086234   -0.012090    0.031278    0.021198
75%      0.633798    0.717495    0.646337    0.716754    0.689467
max      3.484608    2.799842    3.175704    3.333989    3.476817
```

You can select specific percentiles to include in the output:

```
In [73]: series.describe(percentiles=[.05, .25, .75, .95])
Out[73]:
count    500.000000
mean      -0.035852
std        1.017672
min       -3.870073
5%        -1.764398
25%       -0.645819
50%       -0.051329
75%        0.646074
95%        1.663934
max        3.246603
dtype: float64
```

For a non-numerical Series object, `describe()` will give a simple summary of the number of unique values and most frequently occurring values:

```
In [74]: s = pd.Series(['a', 'a', 'b', 'b', 'a', 'a', np.nan, 'c', 'd', 'a'])

In [75]: s.describe()
Out[75]:
count      9
unique     4
top        a
freq       5
dtype: object
```

Note that on a mixed-type DataFrame object, `describe()` will restrict the summary to include only numerical columns or, if none are, only categorical columns:

```
In [76]: frame = pd.DataFrame({'a': ['Yes', 'Yes', 'No', 'No'], 'b': range(4)})

In [77]: frame.describe()
Out[77]:
              b
count  4.000000
mean   1.500000
std    1.290994
min    0.000000
25%    0.750000
50%    1.500000
75%    2.250000
max    3.000000
```

This behaviour can be controlled by providing a list of types as `include/exclude` arguments. The special value all can also be used:

```
In [78]: frame.describe(include=['object'])
Out[78]:
          a
count     4
unique    2
top     Yes
freq      2

In [79]: frame.describe(include=['number'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[79]:
              b
count  4.000000
mean   1.500000
std    1.290994
min    0.000000
25%    0.750000
50%    1.500000
75%    2.250000
max    3.000000

In [80]: frame.describe(include='all')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
          a         b
count     4  4.000000
unique    2       NaN
top     Yes       NaN
freq      2       NaN
mean    NaN  1.500000
std     NaN  1.290994
min     NaN  0.000000
25%     NaN  0.750000
50%     NaN  1.500000
75%     NaN  2.250000
max     NaN  3.000000
```

Index of Min/Max Values

The `idxmin()` and `idxmax()` functions on Series and DataFrame compute the index labels with the minimum and maximum corresponding values:

```
In [81]: s1 = pd.Series(np.random.randn(5))

In [82]: s1
Out[82]:
0    1.481007
1    0.718269
2    0.320711
3   -0.597705
4   -0.140049
dtype: float64

In [83]: s1.idxmin(), s1.idxmax()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[83

In [84]: df1 = pd.DataFrame(np.random.randn(5,3), columns=['A','B','C'])

In [85]: df1
Out[85]:
          A         B         C
0 -0.361656  0.266880  0.783477
1  1.612599  0.557871 -0.232650
2 -0.202611  0.926477 -0.668062
3 -0.883837 -0.350701  0.736130
4 -1.598094  0.824303 -1.996433

In [86]: df1.idxmin(axis=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
A    4
B    3
C    4
dtype: int64

In [87]: df1.idxmin(axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
0    A
1    C
2    C
3    A
4    C
dtype: object
```

When there are multiple rows (or columns) matching the minimum or maximum value, `idxmin()` and `idxmax()` return the first matching index:

```
In [88]: df3 = pd.DataFrame([2, 1, 1, 3, np.nan], columns=['A'], index=list('edcba'))

In [89]: df3
Out[89]:
     A
e  2.0
d  1.0
c  1.0
b  3.0
a  NaN
```

```
In [90]: df3['A'].idxmin()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[90]: 'd'
```

Value counts (histogramming) / Mode

The `value_counts()` Series method and top-level function computes a histogram of a 1D array of values.

```
In [91]: data = np.random.randint(0, 7, size=50)

In [92]: data
Out[92]:
array([0, 0, 0, 1, 1, 5, 3, 3, 2, 3, 0, 0, 1, 4, 4, 6, 6, 0, 2, 4, 2, 0, 3,
       1, 1, 2, 2, 1, 1, 5, 3, 0, 5, 4, 6, 0, 2, 6, 2, 1, 2, 3, 5, 5, 4, 1,
       6, 0, 3, 6])

In [93]: s = pd.Series(data)

In [94]: s.value_counts()
Out[94]:
0    10
1     9
2     8
3     7
6     6
5     5
4     5
dtype: int64
```

Similarly, you can get the most frequently occurring value(s) (the mode) of the values in a Series or DataFrame:

```
In [95]: s5 = pd.Series([1, 1, 3, 3, 3, 5, 5, 7, 7, 7])

In [96]: s5.mode()
Out[96]:
0    3
1    7
dtype: int64

In [97]: df5 = pd.DataFrame({'A': np.random.randint(0, 7, size=50),
   ....:                     'B': np.random.randint(-10, 15, size=50)})
   ....:

In [98]: df5.mode()
Out[98]:
   A  B
0  5  2
```

Discretization and quantiling

Continuous values can be discretized using the `cut()` (bins based on values) and `qcut()` (bins based on sample quantiles) functions:

```
In [99]: arr = np.random.randn(20)

In [100]: factor = pd.cut(arr, 4)
In [101]: factor
Out[101]:
[(-2.0973, -1.174], (0.665, 1.585], (0.665, 1.585], (-0.254, 0.665], (-2.0973, -1.174], ..., (-1.174,
Length: 20
Categories (4, object): [(-2.0973, -1.174] < (-1.174, -0.254] < (-0.254, 0.665] < (0.665, 1.585]]

In [102]: factor = pd.cut(arr, [-5, -1, 0, 1, 5])

In [103]: factor
Out[103]:
[(-5, -1], (1, 5], (1, 5], (0, 1], (-5, -1], ..., (-1, 0], (-5, -1], (-1, 0], (0, 1], (-1, 0]]
Length: 20
Categories (4, object): [(-5, -1] < (-1, 0] < (0, 1] < (1, 5]]
```

`qcut()` computes sample quantiles. For example, we could slice up some normally distributed data into equal-size quartiles like so:

```
In [104]: factor = pd.qcut(arr, [0, .25, .5, .75, 1])

In [105]: factor
Out[105]:
[[-2.0936, -1.134], (-0.181, 1.585], (-0.181, 1.585], (-0.181, 1.585], [-2.0936, -1.134], ..., (-0.66
Length: 20
Categories (4, object): [[-2.0936, -1.134] < (-1.134, -0.663] < (-0.663, -0.181] < (-0.181, 1.585]]

In [106]: pd.value_counts(factor)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
(-0.181, 1.585]      5
(-0.663, -0.181]     5
(-1.134, -0.663]     5
[-2.0936, -1.134]    5
dtype: int64
```

We can also pass infinite values to define the bins:

```
In [107]: arr = np.random.randn(20)

In [108]: factor = pd.cut(arr, [-np.inf, 0, np.inf])

In [109]: factor
Out[109]:
[(0, inf], (0, inf], (0, inf], (0, inf], (-inf, 0], ..., (0, inf], (-inf, 0], (-inf, 0], (-inf, 0],
Length: 20
Categories (2, object): [(-inf, 0] < (0, inf]]
```

Function application

Row or Column-wise Function Application

Arbitrary functions can be applied along the axes of a DataFrame using the `apply()` method, which, like the descriptive statistics methods, take an optional axis argument:

```
In [110]: df = pd.DataFrame({'one' : pd.Series(np.random.randn(3), index=['a', 'b', 'c']),
   .....: 'two' : pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd']),
   .....: 'three' : pd.Series(np.random.randn(3), index=['b', 'c', 'd'])})
   .....:

In [111]: df
Out[111]:
        one     three       two
a -1.268951       NaN  1.268233
b -0.260681  0.431861  1.658656
c -0.853457 -0.877516  0.296434
d       NaN -0.338294  1.029727

In [112]: df.apply(np.mean)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
one     -0.794363
three   -0.261316
two      1.063262
dtype: float64

In [113]: df.apply(np.mean, axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
a   -0.000359
b    0.609945
c   -0.478180
d    0.345716
dtype: float64

In [114]: df.apply(lambda x: x.max() - x.min())
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
one      1.008270
three    1.309377
two      1.362222
dtype: float64

In [115]: df.apply(np.cumsum)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
        one     three       two
a -1.268951       NaN  1.268233
b -1.529632  0.431861  2.926889
c -2.383089 -0.445655  3.223322
d       NaN -0.783949  4.253049
```

Depending on the return type of the function passed to `apply()`, the result will either be of lower dimension or the same dimension.

`apply()` combined with some cleverness can be used to answer many questions about a data set. For example, suppose we wanted to extract the date where the maximum value for each column occurred:

```
In [116]: tsdf = pd.DataFrame(np.random.randn(1000, 3), columns=['A', 'B', 'C'],
   .....: index=pd.date_range('1/1/2000', periods=1000))
   .....:

In [117]: tsdf.apply(lambda x: x.idxmax())
Out[117]:
A   2001-11-24
B   2000-09-07
C   2002-08-10
dtype: datetime64[ns]
```

You may also pass additional arguments and keyword arguments to the `apply()` method. For instance, consider the following function you would like to apply:

```
In [118]: def subtract_and_divide(x, sub, divide=1):
   .....:     return (x - sub) / divide
   .....:

In [119]: df.apply(subtract_and_divide, args=(5,), divide=3)
Out[119]:
        one     three       two
a -2.089650       NaN -1.243922
b -1.753560 -1.522713 -1.113781
c -1.951152 -1.959172 -1.567855
d       NaN -1.779431 -1.323424
```

Another useful feature is the ability to pass Series methods to carry out some Series operation on each column or row:

```
In [120]: tsdf = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
   .....: index=pd.date_range('1/1/2000', periods=10))
   .....:

In [121]: tsdf.ix[4:8] = np.nan

In [122]: tsdf
Out[122]:
                   A         B         C
2000-01-01 -1.701747  0.338606  1.015076
2000-01-02  0.150664  1.668121 -0.828408
2000-01-03 -0.070239 -1.540131 -2.028955
2000-01-04 -0.285152  0.602373 -0.371809
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08       NaN       NaN       NaN
2000-01-09 -0.531292  0.853047 -0.288414
2000-01-10 -0.146233  1.290812 -0.519207

In [123]: tsdf.apply(pd.Series.interpolate)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
                   A         B         C
2000-01-01 -1.701747  0.338606  1.015076
2000-01-02  0.150664  1.668121 -0.828408
2000-01-03 -0.070239 -1.540131 -2.028955
2000-01-04 -0.285152  0.602373 -0.371809
2000-01-05 -0.334380  0.652507 -0.355130
2000-01-06 -0.383608  0.702642 -0.338451
2000-01-07 -0.432836  0.752777 -0.321772
2000-01-08 -0.482064  0.802912 -0.305093
2000-01-09 -0.531292  0.853047 -0.288414
2000-01-10 -0.146233  1.290812 -0.519207
```

Applying elementwise Python functions

Since not all functions can be vectorized (accept NumPy arrays and return another array or value), the methods `applymap()` on DataFrame and analogously `map()` on Series accept any Python function taking a single value and returning a single value. For example:

```
In [124]: df
Out[124]:
        one     three       two
```

```
a -1.268951       NaN  1.268233
b -0.260681  0.431861  1.658656
c -0.853457 -0.877516  0.296434
d       NaN -0.338294  1.029727

In [125]: df['one'].map(lambda x: len(str(x)))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
a    17
b    19
c    19
d     3
Name: one, dtype: int64

In [126]: df.applymap(lambda x: len(str(x)))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
   one  three  two
a   17      3   17
b   19     18   18
c   19     19   17
d    3     20   18
```

Reindexing and altering labels

`reindex()` is the fundamental data alignment method in pandas. It is used to implement nearly all other features relying on label-alignment functionality. To reindex means to conform the data to match a given set of labels along a particular axis. This accomplishes several things:

- Reorders the existing data to match a new set of labels
- Inserts missing value (NA) markers in label locations where no data for that label existed
- If specified, fill data for missing labels using logic (highly relevant to working with time series data)

Here is a simple example:

```
In [127]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [128]: s.reindex(['e', 'b', 'f', 'd'])
Out[128]:
e   -1.970569
b    1.275068
f         NaN
d   -1.093001
dtype: float64
```

With a DataFrame, you can simultaneously reindex the index and columns:

```
In [129]: df
Out[129]:
        one     three       two
a -1.268951       NaN  1.268233
b -0.260681  0.431861  1.658656
c -0.853457 -0.877516  0.296434
d       NaN -0.338294  1.029727

In [130]: df.reindex(index=['c', 'f', 'b'], columns=['three', 'two', 'one'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
      three       two       one
```

```
c -0.877516  0.296434 -0.853457
f       NaN       NaN       NaN
b  0.431861  1.658656 -0.260681
```

Reindexing to align with another object

You may wish to take an object and reindex its axes to be labeled the same as another object.

```
In [131]: df.reindex_like(df.ix[:2, 2:])
Out[131]:
        two
a  1.268233
b  1.658656
```

Aligning objects with each other with `align`

The `align()` method is the fastest way to simultaneously align two objects. It supports a `join` argument (related to joining and merging):

- `join='outer'`: take the union of the indexes (default)
- `join='left'`: use the calling object's index
- `join='right'`: use the passed object's index
- `join='inner'`: intersect the indexes

It returns a tuple with both of the reindexed Series:

```
In [132]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [133]: s1 = s[:4]

In [134]: s2 = s[1:]

In [135]: s1.align(s2)
Out[135]:
(a   -0.917421
 b    1.500971
 c    2.170547
 d    0.066315
 e         NaN
 dtype: float64, a         NaN
 b    1.500971
 c    2.170547
 d    0.066315
 e   -0.310747
 dtype: float64)

In [136]: s1.align(s2, join='inner')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
(b    1.500971
 c    2.170547
 d    0.066315
 dtype: float64, b    1.500971
 c    2.170547
 d    0.066315
 dtype: float64)

In [137]: s1.align(s2, join='left')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
(a   -0.917421
 b    1.500971
 c    2.170547
 d    0.066315
 dtype: float64, a         NaN
 b    1.500971
 c    2.170547
 d    0.066315
 dtype: float64)
```

For DataFrames, the join method will be applied to both the index and the columns by default:

```
In [138]: df = pd.DataFrame({'one' : pd.Series(np.random.randn(3), index=['a', 'b', 'c']),
   .....: 'two' : pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd']),
   .....: 'three' : pd.Series(np.random.randn(3), index=['b', 'c', 'd'])})
   .....:

In [139]: df2 = pd.DataFrame({'two' : pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'e']),
   .....: 'three' : pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd'])})
   .....:

In [140]:

In [140]: df2
Out[140]:
      three       two
a  1.213021 -1.528701
b  0.763892  2.782527
c  0.229913  1.103515
d  0.433320       NaN
e       NaN -0.596160

In [141]: df.align(df2, join='inner')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
(       three       two
 a        NaN  0.353936
 b -0.126721 -0.233720
 c -1.367344 -0.396185
 d -0.665558  0.235321,       three       two
 a  1.213021 -1.528701
 b  0.763892  2.782527
 c  0.229913  1.103515
 d  0.433320       NaN)
```

You can also pass an axis option to only align on the specified `axis`:

```
In [142]: df.align(df2, join='inner', axis=0)
Out[142]:
(        one     three       two
 a -1.380342       NaN  0.353936
 b -0.710816 -0.126721 -0.233720
 c  0.213463 -1.367344 -0.396185
 d       NaN -0.665558  0.235321,       three       two
 a  1.213021 -1.528701
 b  0.763892  2.782527
 c  0.229913  1.103515
 d  0.433320       NaN)
```

Filling while reindexing

`reindex()` takes an optional parameter method which is a filling method chosen from the following options:

- pad / ffill: Fill values forward
- bfill / backfill: Fill values backward
- nearest: Fill from the nearest index value

These methods require that the indexes are **ordered** increasing or decreasing.

We illustrate these fill methods on a simple Series:

```
In [143]: rng = pd.date_range('1/3/2000', periods=8)

In [144]: ts = pd.Series(np.random.randn(8), index=rng)

In [145]: ts2 = ts[[0, 3, 6]]

In [146]: ts
Out[146]: 
2000-01-03    0.276845
2000-01-04   -0.079263
2000-01-05    1.299173
2000-01-06   -1.027178
2000-01-07   -2.024731
2000-01-08   -1.881748
2000-01-09    1.553641
2000-01-10   -0.541811
Freq: D, dtype: float64

In [147]: ts2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
2000-01-03    0.276845
2000-01-06   -1.027178
2000-01-09    1.553641
dtype: float64

In [148]: ts2.reindex(ts.index)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
2000-01-03    0.276845
2000-01-04         NaN
2000-01-05         NaN
2000-01-06   -1.027178
2000-01-07         NaN
2000-01-08         NaN
2000-01-09    1.553641
2000-01-10         NaN
Freq: D, dtype: float64

In [149]: ts2.reindex(ts.index, method='ffill')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
2000-01-03    0.276845
2000-01-04    0.276845
2000-01-05    0.276845
```

```
2000-01-06   -1.027178
2000-01-07   -1.027178
2000-01-08   -1.027178
2000-01-09    1.553641
2000-01-10    1.553641
Freq: D, dtype: float64

In [150]: ts2.reindex(ts.index, method='bfill')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
2000-01-03    0.276845
2000-01-04   -1.027178
2000-01-05   -1.027178
2000-01-06   -1.027178
2000-01-07    1.553641
2000-01-08    1.553641
2000-01-09    1.553641
2000-01-10         NaN
Freq: D, dtype: float64

In [151]: ts2.reindex(ts.index, method='nearest')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
2000-01-03    0.276845
2000-01-04    0.276845
2000-01-05   -1.027178
2000-01-06   -1.027178
2000-01-07   -1.027178
2000-01-08    1.553641
2000-01-09    1.553641
2000-01-10    1.553641
Freq: D, dtype: float64
```

Dropping labels from an axis

A method closely related to reindex is the `drop()` function. It removes a set of labels from an axis:

```
In [152]: df
Out[152]:
        one     three       two
a -1.380342       NaN  0.353936
b -0.710816 -0.126721 -0.233720
c  0.213463 -1.367344 -0.396185
d       NaN -0.665558  0.235321

In [153]: df.drop(['a', 'd'], axis=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
        one     three       two
b -0.710816 -0.126721 -0.233720
c  0.213463 -1.367344 -0.396185

In [154]: df.drop(['one'], axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
      three       two
a       NaN  0.353936
b -0.126721 -0.233720
c -1.367344 -0.396185
d -0.665558  0.235321
```

Renaming / mapping labels

The `rename()` method allows you to relabel an axis based on some mapping (a dict or Series) or an arbitrary function.

```
In [155]: s
Out[155]:
a   -0.917421
b    1.500971
c    2.170547
d    0.066315
e   -0.310747
dtype: float64

In [156]: s.rename(str.upper)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[1
A   -0.917421
B    1.500971
C    2.170547
D    0.066315
E   -0.310747
dtype: float64
```

If you pass a function, it must return a value when called with any of the labels (and must produce a set of unique values). But if you pass a dict or Series, it need only contain a subset of the labels as keys:

```
In [157]: df.rename(columns={'one' : 'foo', 'two' : 'bar'},
   .....: index={'a' : 'apple', 'b' : 'banana', 'd' : 'durian'})
   .....:
Out[157]:
             foo     three       bar
apple  -1.380342       NaN  0.353936
banana -0.710816 -0.126721 -0.233720
c       0.213463 -1.367344 -0.396185
durian       NaN -0.665558  0.235321
```

The `rename()` method also provides an `inplace` named parameter that is by default False and copies the underlying data. Pass `inplace=True` to rename the data in place.

Sorting by index and value

There are two obvious kinds of sorting that you may be interested in: sorting by label and sorting by actual values. The primary method for sorting axis labels (indexes) across data structures is the `sort_index()` method.

```
In [158]: unsorted_df = df.reindex(index=['a', 'd', 'c', 'b'],
   .....: columns=['three', 'two', 'one'])
   .....:

In [159]: unsorted_df.sort_index()
Out[159]:
      three       two       one
a       NaN  0.353936 -1.380342
b -0.126721 -0.233720 -0.710816
c -1.367344 -0.396185  0.213463
d -0.665558  0.235321       NaN

In [160]: unsorted_df.sort_index(ascending=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
      three       two       one
d -0.665558  0.235321       NaN
c -1.367344 -0.396185  0.213463
b -0.126721 -0.233720 -0.710816
a       NaN  0.353936 -1.380342

In [161]: unsorted_df.sort_index(axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
        one     three       two
a -1.380342       NaN  0.353936
d       NaN -0.665558  0.235321
c  0.213463 -1.367344 -0.396185
b -0.710816 -0.126721 -0.233720
```

`DataFrame.sort_index()` can accept an optional `by` argument for `axis=0` which will use an arbitrary vector or a column name of the DataFrame to determine the sort order:

```
In [162]: df1 = pd.DataFrame({'one':[2,1,1,1],'two':[1,3,2,4],'three':[5,4,3,2]})

In [163]: df1.sort_index(by='two')
Out[163]:
   one  three  two
0    2      5    1
2    1      3    2
1    1      4    3
3    1      2    4
```

The by argument can take a list of column names, e.g.:

```
In [164]: df1[['one', 'two', 'three']].sort_index(by=['one','two'])
Out[164]:
   one  two  three
2    1    2      3
1    1    3      4
3    1    4      2
0    2    1      5
```

Smallest / largest values

Series has the `nsmallest()` and `nlargest()` methods which return the smallest or largest n values. For a large Series this can be much faster than sorting the entire Series and calling `head(n)` on the result.

```
In [165]: s = pd.Series(np.random.permutation(10))

In [166]: s
Out[166]:
0    0
1    9
2    5
3    3
4    8
5    7
6    2
7    1
8    6
9    4
dtype: int64
```

```
In [167]: s.order()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[16
0    0
7    1
6    2
3    3
9    4
2    5
8    6
5    7
4    8
1    9
dtype: int64

In [168]: s.nsmallest(3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
0    0
7    1
6    2
dtype: int64

In [169]: s.nlargest(3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
1    9
4    8
5    7
dtype: int64
```

Sorting by a multi-index column

You must be explicit about sorting when the column is a multi-index, and fully specify all levels to `by`.

```
In [170]: df1.columns = pd.MultiIndex.from_tuples([('a','one'),('a','two'),('b','three')])

In [171]: df1.sort_index(by=('a','two'))
Out[171]:
    a         b
  one two three
3   1   2     4
2   1   3     2
1   1   4     3
0   2   5     1
```

Indexing and selecting data

Different Choices for Indexing

Pandas supports three types of multi-axis indexing.

- `.loc` is primarily label based, but may also be used with a boolean array. `.loc` will raise `KeyError` when the items are not found. Allowed inputs are:
  - A single label, e.g. `5` or `'a'`, (note that `5` is interpreted as a *label* of the index. This use is not an integer position along the index)

   - A list or array of labels `['a', 'b', 'c']`
   - A slice object with labels `'a':'f'`, (note that contrary to usual python slices, both the start and the stop are included!)
   - A boolean array

- `.iloc` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array. `.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except slice indexers which allow out-of-bounds indexing. Allowed inputs are:
   - An integer e.g. `5`
   - A list or array of integers `[4, 3, 0]`
   - A slice object with ints `1:7`
   - A boolean array

- `.ix` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type. `.ix` is the most general and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierachical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

Selection By Position

A few basic examples:

```
In [172]: s1 = pd.Series(np.random.randn(5),index=list(range(0,10,2)))

In [173]: s1
Out[173]:
0    2.084884
2   -0.124936
4   -0.432126
6   -0.204394
8   -0.298929
dtype: float64

In [174]: s1.iloc[:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[1
0    2.084884
2   -0.124936
4   -0.432126
dtype: float64

In [175]: s1.iloc[3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

In [176]: s1.iloc[:3] = 0

In [177]: s1
Out[177]:
0    0.000000
2    0.000000
4    0.000000
6   -0.204394
```

```
8   -0.298929
dtype: float64
```

With a DataFrame:

```
In [178]: df1 = pd.DataFrame(np.random.randn(6,4),
   .....:                    index=list(range(0,12,2)),
   .....:                    columns=list(range(0,8,2)))
   .....:

In [179]: df1
Out[179]:
           0         2         4         6
0  -0.727100  1.097008  0.322402 -0.373036
2  -0.346949 -0.918235  1.271646 -0.097125
4   0.354153 -0.150776  0.517917  0.838237
6   1.415195  0.218133  0.604036 -0.536193
8  -1.146182 -0.437254 -1.297848  0.835153
10  1.299792 -0.579843 -1.740963 -0.065243

In [180]: df1.iloc[:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
          0         2         4         6
0 -0.727100  1.097008  0.322402 -0.373036
2 -0.346949 -0.918235  1.271646 -0.097125
4  0.354153 -0.150776  0.517917  0.838237

In [181]: df1.iloc[1:5, 2:4]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
          4         6
2  1.271646 -0.097125
4  0.517917  0.838237
6  0.604036 -0.536193
8 -1.297848  0.835153

In [182]: df1.iloc[[1, 3, 5], [1, 3]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
           2         6
2  -0.918235 -0.097125
6   0.218133 -0.536193
10 -0.579843 -0.065243

In [183]: df1.iloc[1:3, :]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
          0         2         4         6
2 -0.346949 -0.918235  1.271646 -0.097125
4  0.354153 -0.150776  0.517917  0.838237

In [184]: df1.iloc[:, 1:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
           2         4
0   1.097008  0.322402
2  -0.918235  1.271646
4  -0.150776  0.517917
6   0.218133  0.604036
8  -0.437254 -1.297848
10 -0.579843 -1.740963
```

```
In [185]: df1.iloc[1, 1]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

In [186]: df1.iloc[1]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
0   -0.346949
2   -0.918235
4    1.271646
6   -0.097125
Name: 2, dtype: float64
```

Boolean indexing

Another common operation is the use of boolean vectors to filter the data. The operators are: | for or, & for and, and ~ for not. These must be grouped by using parentheses.

Using a boolean vector to index a Series works exactly as in a numpy ndarray:

```
In [187]: s = pd.Series(range(-3, 4))

In [188]: s
Out[188]:
0   -3
1   -2
2   -1
3    0
4    1
5    2
6    3
dtype: int64

In [189]: s[s > 0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[189]:
4    1
5    2
6    3
dtype: int64

In [190]: s[(s < -1) | (s > 0.5)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
0   -3
1   -2
4    1
5    2
6    3
dtype: int64

In [191]: s[~(s < 0)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
3    0
4    1
5    2
6    3
dtype: int64
```

You may select rows from a DataFrame using a boolean vector the same length as the DataFrame's index (for example, something derived from one of the columns of the DataFrame):

```
In [192]: df = pd.DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
   .....:                    'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
   .....:                    'c' : np.random.randn(7)})
   .....:

In [193]: df
Out[193]:
       a  b         c
0    one  x -1.667398
1    one  y -0.839763
2    two  y -0.431709
3  three  x  0.674604
4    two  y -0.017719
5    one  x  0.640781
6    six  x -1.659088

In [194]: df[df['c'] > 0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
       a  b         c
3  three  x  0.674604
5    one  x  0.640781

In [195]: criterion = df['a'].map(lambda x: x.startswith('t'))

In [196]: df[criterion]
Out[196]:
       a  b         c
2    two  y -0.431709
3  three  x  0.674604
4    two  y -0.017719

In [197]: df[criterion & (df['b'] == 'x')]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Ou
       a  b         c
3  three  x  0.674604

In [198]: df.loc[criterion & (df['b'] == 'x'), 'b':'c']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
   b         c
3  x  0.674604
```

Indexing with isin

Consider the isin method of Series, which returns a boolean vector that is true wherever the Series elements exist in the passed list. This allows you to select rows where one or more columns have values you want:

```
In [199]: s = pd.Series(np.arange(5), index=np.arange(5)[::-1])

In [200]: s
Out[200]:
4    0
3    1
2    2
1    3
0    4
dtype: int64
```

```
In [201]: s.isin([2, 4, 6])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[201]:
4    False
3    False
2     True
1    False
0     True
dtype: bool

In [202]: s[s.isin([2, 4, 6])]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
2    2
0    4
dtype: int64
```

The same method is available for `Index` objects and is useful for the cases when you don't know which of the sought labels are in fact present:

```
In [203]: s[s.index.isin([2, 4, 6])]
Out[203]:
4    0
2    2
dtype: int64

In [204]: s[[2, 4, 6]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[204]:
2    2.0
4    0.0
6    NaN
dtype: float64
```

In addition to that, `MultiIndex` allows selecting a separate level to use in the membership check:

```
In [205]: s_mi = pd.Series(np.arange(6),
   .....: index=pd.MultiIndex.from_product([[0, 1], ['a', 'b', 'c']]))
   .....:

In [206]: s_mi
Out[206]:
0  a    0
   b    1
   c    2
1  a    3
   b    4
   c    5
dtype: int64

In [207]: s_mi.iloc[s_mi.index.isin([(1, 'a'), (2, 'b'), (0, 'c')])]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[207]:
0  c    2
1  a    3
dtype: int64

In [208]: s_mi.iloc[s_mi.index.isin(['a', 'c', 'e'], level=1)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
0  a    0
   c    2
1  a    3
   c    5
```

```
dtype: int64
```

DataFrame also has an `isin` method. When calling `isin`, pass a set of values as either an array or dict. If values is an array, `isin` returns a DataFrame of booleans that is the same shape as the original DataFrame, with True wherever the element is in the sequence of values.

```
In [209]: df = pd.DataFrame({'vals': [1, 2, 3, 4], 'ids': ['a', 'b', 'f', 'n'],
   .....: 'ids2': ['a', 'n', 'c', 'n']})
   .....:

In [210]: df
Out[210]:
  ids ids2  vals
0   a    a     1
1   b    n     2
2   f    c     3
3   n    n     4

In [211]: df.isin(['a', 'b', 1, 3])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[2
     ids   ids2   vals
0   True   True   True
1   True  False  False
2  False  False   True
3  False  False  False

In [212]: df.isin({'ids': ['a', 'b'], 'vals': [1, 3]})
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
     ids   ids2   vals
0   True  False   True
1   True  False  False
2  False  False   True
3  False  False  False
```

Set / Reset Index

DataFrame has a `set_index` method which takes a column name (for a regular `Index`) or a list of column names (for a `MultiIndex`), to create a new, indexed DataFrame:

```
In [213]: data = pd.DataFrame({'a' : ['bar', 'bar', 'foo', 'foo'],
   .....:                      'b' : ['one', 'two', 'one', 'two'],
   .....:                      'c' : ['z', 'y', 'x', 'w'],
   .....:                      'd' : range(1, 5)})
   .....:

In [214]: data
Out[214]:
     a    b  c  d
0  bar  one  z  1
1  bar  two  y  2
2  foo  one  x  3
3  foo  two  w  4

In [215]: data.set_index('c')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
     a    b  d
c
```

```
z  bar  one  1
y  bar  two  2
x  foo  one  3
w  foo  two  4

In [216]: data.set_index(['a', 'b'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
         c  d
a   b
bar one  z  1
    two  y  2
foo one  x  3
    two  w  4

In [217]: data.set_index(['a', 'b'], inplace=True)
```

`reset_index` is the inverse operation to `set_index`.

```
In [218]: data.reset_index()
Out[218]:
     a    b  c  d
0  bar  one  z  1
1  bar  two  y  2
2  foo  one  x  3
3  foo  two  w  4

In [219]: data.reset_index(level='a')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
       a  c  d
b
one  bar  z  1
two  bar  y  2
one  foo  x  3
two  foo  w  4
```

Todo

Complete **Pandas** section

CHAPTER 5

Data I/O

Source: http://pandas.pydata.org/pandas-docs/stable/io.html

Todo

Write **Data I/O** section

Data import

From CSV

From Excel

From Stata

From MatLab

From Web

Yahoo! Finance

Google Finance

FRED

Fama/French

World Bank

Google Analytics

Quandl

From html

From unstructured files

Data export

To HDF

To CSV

To Excel

To Stata

CHAPTER 6

Data crunching examples

CHAPTER 7

Data visualization

Todo

Write **Data visualization** section

Matplotlib

- A few examples

Seaborn

Bokeh

Plotly

CHAPTER 8

What’s missing

Todo

Write **Data I/O** section

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/dataanalysispython/checkouts/latest/notes/dataio.rst, line 7.)

Todo

Complete **Pandas** section

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/dataanalysispython/checkouts/latest/notes/pandas.rst line 1841.)

Todo

Write **Data visualization** section

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/dataanalysispython/checkouts/latest/notes/visualizati line 5.)