**Pandas basics**

*Source: https://pandas.pydata.org/docs/user\_guide/basics.html*

Essential basic functionality pandas 2.3.0 documentation Skip to main content Back to top CtrlK Site Navigation Getting started User Guide API reference Development Release notes GitHub Twitter Mastodon Site Navigation Getting started User Guide API reference Development Release notes GitHub Twitter Mastodon 10 minutes to pandas Intro to data structures Essential basic functionality IO tools (text, CSV, HDF5, ) PyArrow Functionality Indexing and selecting data MultiIndex advanced indexing Copy-on-Write (CoW) Merge, join, concatenate and compare Reshaping and pivot tables Working with text data Working with missing data Duplicate Labels Categorical data Nullable integer data type Nullable Boolean data type Chart visualization Table Visualization Group by: split-apply-combine Windowing operations Time series date functionality Time deltas Options and settings Enhancing performance Scaling to large datasets Sparse data structures Frequently Asked Questions (FAQ) Cookbook User Guide Essential... Essential basic functionality Here we discuss a lot of the essential functionality common to the pandas data structures. To begin, lets create some example objects like we did in the 10 minutes to pandas section: In [1]: index pd.date\_range(112000, periods8) In [2]: s pd.Series(np.random.randn(5), index[a, b, c, d, e]) In [3]: df pd.DataFrame(np.random.randn(8, 3), indexindex, 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 [4]: long\_series pd.Series(np.random.randn(1000)) In [5]: long\_series.head() Out[5]: 0 -1.157892 1 -1.344312 2 0.844885 3 1.075770 4 -0.109050 dtype: float64 In [6]: long\_series.tail(3) Out[6]: 997 -0.289388 998 -1.020544 999 0.589993 dtype: float64 Attributes and underlying data 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 [7]: df[:2] Out[7]: A B C 2000-01-01 -0.173215 0.119209 -1.044236 2000-01-02 -0.861849 -2.104569 -0.494929 In [8]: df.columns [x.lower() for x in df.columns] In [9]: df Out[9]: a b c 2000-01-01 -0.173215 0.119209 -1.044236 2000-01-02 -0.861849 -2.104569 -0.494929 2000-01-03 1.071804 0.721555 -0.706771 2000-01-04 -1.039575 0.271860 -0.424972 2000-01-05 0.567020 0.276232 -1.087401 2000-01-06 -0.673690 0.113648 -1.478427 2000-01-07 0.524988 0.404705 0.577046 2000-01-08 -1.715002 -1.039268 -0.370647 pandas objects (Index, Series, DataFrame) can be thought of as containers for arrays, which hold the actual data and do the actual computation. For many types, the underlying array is a numpy.ndarray. However, pandas and 3rd party libraries may extend NumPys type system to add support for custom arrays (see dtypes). To get the actual data inside a Index or Series, use the .array property In [10]: s.array Out[10]: NumpyExtensionArray [ 0.4691122999071863, -0.2828633443286633, -1.5090585031735124, -1.1356323710171934, 1.2121120250208506] Length: 5, dtype: float64 In [11]: s.index.array Out[11]: NumpyExtensionArray [a, b, c, d, e] Length: 5, dtype: object array will always be an ExtensionArray. The exact details of what an ExtensionArray is and why pandas uses them are a bit beyond the scope of this introduction. See dtypes for more. If you know you need a NumPy array, use to\_numpy() or numpy.asarray(). In [12]: s.to\_numpy() Out[12]: array([ 0.4691, -0.2829, -1.5091, -1.1356, 1.2121]) In [13]: np.asarray(s) Out[13]: array([ 0.4691, -0.2829, -1.5091, -1.1356, 1.2121]) When the Series or Index is backed by an ExtensionArray, to\_numpy() may involve copying data and coercing values. See dtypes for more. to\_numpy() gives some control over the dtype of the resulting numpy.ndarray. For example, consider datetimes with timezones. NumPy doesnt have a dtype to represent timezone-aware datetimes, so there are two possibly useful representations: An object-dtype numpy.ndarray with Timestamp objects, each with the correct tz A datetime64[ns] -dtype numpy.ndarray, where the values have been converted to UTC and the timezone discarded Timezones may be preserved with dtypeobject In [14]: ser pd.Series(pd.date\_range(2000, periods2, tzCET)) In [15]: ser.to\_numpy(dtypeobject) Out[15]: array([Timestamp(2000-01-01 00:00:000100, tzCET), Timestamp(2000-01-02 00:00:000100, tzCET)], dtypeobject) Or thrown away with dtypedatetime64[ns] In [16]: ser.to\_numpy(dtypedatetime64[ns]) Out[16]: array([1999-12-31T23:00:00.000000000, 2000-01-01T23:00:00.000000000], dtypedatetime64[ns]) Getting the raw data inside a DataFrame is possibly a bit more complex. When your DataFrame only has a single data type for all the columns, DataFrame.to\_numpy() will return the underlying data: In [17]: df.to\_numpy() Out[17]: array([[-0.1732, 0.1192, -1.0442], [-0.8618, -2.1046, -0.4949], [ 1.0718, 0.7216, -0.7068], [-1.0396, 0.2719, -0.425 ], [ 0.567 , 0.2762, -1.0874], [-0.6737, 0.1136, -1.4784], [ 0.525 , 0.4047, 0.577 ], [-1.715 , -1.0393, -0.3706]]) If a DataFrame contains homogeneously-typed data, the ndarray can actually be modified in-place, and the changes will be reflected in the data structure. For heterogeneous data (e.g. some of the DataFrames columns are not all the same dtype), this will not be the case. The values attribute itself, unlike the axis labels, cannot be assigned to. Note When working with heterogeneous data, the dtype of the resulting ndarray will be chosen to accommodate all of the data involved. For example, if strings are involved, the result will be of object dtype. If there are only floats and integers, the resulting array will be of float dtype. In the past, pandas recommended Series.values or DataFrame.values for extracting the data from a Series or DataFrame. Youll still find references to these in old code bases and online. Going forward, we recommend avoiding .values and using .array or .to\_numpy(). .values has the following drawbacks: When your Series contains an extension type, its unclear whether Series.values returns a NumPy array or the extension array. Series.array will always return an ExtensionArray, and will never copy data. Series.to\_numpy() will always return a NumPy array, potentially at the cost of copying coercing values. When your DataFrame contains a mixture of data types, DataFrame.values may involve copying data and coercing values to a common dtype, a relatively expensive operation. DataFrame.to\_numpy(), being a method, makes it clearer that the returned NumPy array may not be a view on the same data in the DataFrame. Accelerated operations pandas has support for accelerating certain types of binary numerical and boolean operations using the numexpr library and the bottleneck libraries. These libraries are especially useful when dealing with large data sets, and provide large speedups. numexpr uses smart chunking, caching, and multiple cores. bottleneck is a set of specialized cython routines that are especially fast when dealing with arrays that have nans. Here is a sample (using 100 column x 100,000 row DataFrames): Operation 0.11.0 (ms) Prior Version (ms) Ratio to Prior df1 df2 13.32 125.35 0.1063 df1 df2 21.71 36.63 0.5928 df1 df2 22.04 36.50 0.6039 You are highly encouraged to install both libraries. See the section Recommended Dependencies for more installation info. These are both enabled to be used by default, you can control this by setting the options: pd.set\_option(compute.use\_bottleneck, False) pd.set\_option(compute.use\_numexpr, False) Flexible binary operations With binary operations between pandas data structures, there are two key points of interest: Broadcasting behavior between higher- (e.g. DataFrame) and lower-dimensional (e.g. Series) objects. Missing data in computations. We will demonstrate how to manage these issues independently, though they can be handled simultaneously. Matching broadcasting behavior DataFrame has the methods add(), sub(), mul(), div() and related functions radd(), rsub(), for carrying out binary operations. For broadcasting behavior, Series input is of primary interest. Using these functions, you can use to either match on the index or columns via the axis keyword: In [18]: 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 [19]: df Out[19]: one two three a 1.394981 1.772517 NaN b 0.343054 1.912123 -0.050390 c 0.695246 1.478369 1.227435 d NaN 0.279344 -0.613172 In [20]: row df.iloc[1] In [21]: column df[two] In [22]: df.sub(row, axiscolumns) Out[22]: one two three a 1.051928 -0.139606 NaN b 0.000000 0.000000 0.000000 c 0.352192 -0.433754 1.277825 d NaN -1.632779 -0.562782 In [23]: df.sub(row, axis1) Out[23]: one two three a 1.051928 -0.139606 NaN b 0.000000 0.000000 0.000000 c 0.352192 -0.433754 1.277825 d NaN -1.632779 -0.562782 In [24]: df.sub(column, axisindex) Out[24]: one two three a -0.377535 0.0 NaN b -1.569069 0.0 -1.962513 c -0.783123 0.0 -0.250933 d NaN 0.0 -0.892516 In [25]: df.sub(column, axis0) Out[25]: one two three a -0.377535 0.0 NaN b -1.569069 0.0 -1.962513 c -0.783123 0.0 -0.250933 d NaN 0.0 -0.892516 Furthermore you can align a level of a MultiIndexed DataFrame with a Series. In [26]: dfmi df.copy() In [27]: dfmi.index pd.MultiIndex.from\_tuples( ....: [(1, a), (1, b), (1, c), (2, a)], names[first, second] ....: ) ....: In [28]: dfmi.sub(column, axis0, levelsecond) Out[28]: one two three first second 1 a -0.377535 0.000000 NaN b -1.569069 0.000000 -1.962513 c -0.783123 0.000000 -0.250933 2 a NaN -1.493173 -2.385688 Series and Index also support the divmod() builtin. This function takes the floor division and modulo operation at the same time returning a two-tuple of the same type as the left hand side. For example: In [29]: s pd.Series(np.arange(10)) In [30]: s Out[30]: 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 dtype: int64 In [31]: div, rem divmod(s, 3) In [32]: div Out[32]: 0 0 1 0 2 0 3 1 4 1 5 1 6 2 7 2 8 2 9 3 dtype: int64 In [33]: rem Out[33]: 0 0 1 1 2 2 3 0 4 1 5 2 6 0 7 1 8 2 9 0 dtype: int64 In [34]: idx pd.Index(np.arange(10)) In [35]: idx Out[35]: Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtypeint64) In [36]: div, rem divmod(idx, 3) In [37]: div Out[37]: Index([0, 0, 0, 1, 1, 1, 2, 2, 2, 3], dtypeint64) In [38]: rem Out[38]: Index([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], dtypeint64) We can also do elementwise divmod(): In [39]: div, rem divmod(s, [2, 2, 3, 3, 4, 4, 5, 5, 6, 6]) In [40]: div Out[40]: 0 0 1 0 2 0 3 1 4 1 5 1 6 1 7 1 8 1 9 1 dtype: int64 In [41]: rem Out[41]: 0 0 1 1 2 2 3 0 4 0 5 1 6 1 7 2 8 2 9 3 dtype: int64 Missing data operations with fill values In Series and DataFrame, the arithmetic functions have the option of inputting a fill\_value, namely a value to substitute when at most one of the values at a location are missing. For example, when adding two DataFrame objects, you may wish to treat NaN as 0 unless both DataFrames are missing that value, in which case the result will be NaN (you can later replace NaN with some other value using fillna if you wish). In [42]: df2 df.copy() In [43]: df2.loc[a, three] 1.0 In [44]: df Out[44]: one two three a 1.394981 1.772517 NaN b 0.343054 1.912123 -0.050390 c 0.695246 1.478369 1.227435 d NaN 0.279344 -0.613172 In [45]: df2 Out[45]: one two three a 1.394981 1.772517 1.000000 b 0.343054 1.912123 -0.050390 c 0.695246 1.478369 1.227435 d NaN 0.279344 -0.613172 In [46]: df df2 Out[46]: one two three a 2.789963 3.545034 NaN b 0.686107 3.824246 -0.100780 c 1.390491 2.956737 2.454870 d NaN 0.558688 -1.226343 In [47]: df.add(df2, fill\_value0) Out[47]: one two three a 2.789963 3.545034 1.000000 b 0.686107 3.824246 -0.100780 c 1.390491 2.956737 2.454870 d NaN 0.558688 -1.226343 Flexible comparisons Series and DataFrame have the binary comparison methods eq, ne, lt, gt, le, and ge whose behavior is analogous to the binary arithmetic operations described above: In [48]: df.gt(df2) Out[48]: one two three a False False False b False False False c False False False d False False False In [49]: df2.ne(df) Out[49]: one two three a False False True b False False False c False False False d True False False These operations produce a pandas object of the same type as the left-hand-side input that is of dtype bool. These boolean objects can be used in indexing operations, see the section on Boolean indexing. Boolean reductions You can apply the reductions: empty, any(), all(), and bool() to provide a way to summarize a boolean result. In [50]: (df 0).all() Out[50]: one False two True three False dtype: bool In [51]: (df 0).any() Out[51]: one True two True three True dtype: bool You can reduce to a final boolean value. In [52]: (df 0).any().any() Out[52]: True You can test if a pandas object is empty, via the empty property. In [53]: df.empty Out[53]: False In [54]: pd.DataFrame(columnslist(ABC)).empty Out[54]: True Warning Asserting the truthiness of a pandas object will raise an error, as the testing of the emptiness or values is ambiguous. In [55]: if df: ....: print(True) ....: --------------------------------------------------------------------------- ValueError Traceback (most recent call last) ipython-input-55-318d08b2571a in ?() ---- 1 if df: 2 print(True) workpandaspandaspandascoregeneric.py in ?(self) 1575 final 1576 def \_\_nonzero\_\_(self) - NoReturn: - 1577 raise ValueError( 1578 fThe truth value of a {type(self).\_\_name\_\_} is ambiguous. 1579 Use a.empty, a.bool(), a.item(), a.any() or a.all(). 1580 ) ValueError: The truth value of a DataFrame is ambiguous. Use a.empty, a.bool(), a.item(), a.any() or a.all(). In [56]: df and df2 --------------------------------------------------------------------------- ValueError Traceback (most recent call last) ipython-input-56-b241b64bb471 in ?() ---- 1 df and df2 workpandaspandaspandascoregeneric.py in ?(self) 1575 final 1576 def \_\_nonzero\_\_(self) - NoReturn: - 1577 raise ValueError( 1578 fThe truth value of a {type(self).\_\_name\_\_} is ambiguous. 1579 Use a.empty, a.bool(), a.item(), a.any() or a.all(). 1580 ) ValueError: The truth value of a DataFrame is ambiguous. Use a.empty, a.bool(), a.item(), a.any() or a.all(). See gotchas for a more detailed discussion. Comparing if objects are equivalent Often you may find that there is more than one way to compute the same result. As a simple example, consider df df and df 2. To test that these two computations produce the same result, given the tools shown above, you might imagine using (df df df 2).all(). But in fact, this expression is False: In [57]: df df df 2 Out[57]: one two three a True True False b True True True c True True True d False True True In [58]: (df df df 2).all() Out[58]: one False two True three False dtype: bool Notice that the boolean DataFrame df df df 2 contains some False values! This is because NaNs do not compare as equals: In [59]: np.nan np.nan Out[59]: False So, NDFrames (such as Series and DataFrames) have an equals() method for testing equality, with NaNs in corresponding locations treated as equal. In [60]: (df df).equals(df 2) Out[60]: True Note that the Series or DataFrame index needs to be in the same order for equality to be True: In [61]: df1 pd.DataFrame({col: [foo, 0, np.nan]}) In [62]: df2 pd.DataFrame({col: [np.nan, 0, foo]}, index[2, 1, 0]) In [63]: df1.equals(df2) Out[63]: False In [64]: df1.equals(df2.sort\_index()) Out[64]: True Comparing array-like objects You can conveniently perform element-wise comparisons when comparing a pandas data structure with a scalar value: In [65]: pd.Series([foo, bar, baz]) foo Out[65]: 0 True 1 False 2 False dtype: bool In [66]: pd.Index([foo, bar, baz]) foo Out[66]: array([ True, False, False]) pandas also handles element-wise comparisons between different array-like objects of the same length: In [67]: pd.Series([foo, bar, baz]) pd.Index([foo, bar, qux]) Out[67]: 0 True 1 True 2 False dtype: bool In [68]: pd.Series([foo, bar, baz]) np.array([foo, bar, qux]) Out[68]: 0 True 1 True 2 False dtype: bool Trying to compare Index or Series objects of different lengths will raise a ValueError: In [69]: pd.Series([foo, bar, baz]) pd.Series([foo, bar]) --------------------------------------------------------------------------- ValueError Traceback (most recent call last) Cell In[69], line 1 ---- 1 pd.Series([foo, bar, baz]) pd.Series([foo, bar]) File workpandaspandaspandascoreopscommon.py:76, in \_unpack\_zerodim\_and\_defer.locals.new\_method(self, other) 72 return NotImplemented 74 other item\_from\_zerodim(other) --- 76 return method(self, other) File workpandaspandaspandascorearraylike.py:40, in OpsMixin.\_\_eq\_\_(self, other) 38 unpack\_zerodim\_and\_defer(\_\_eq\_\_) 39 def \_\_eq\_\_(self, other): --- 40 return self.\_cmp\_method(other, operator.eq) File workpandaspandaspandascoreseries.py:6125, in Series.\_cmp\_method(self, other, op) 6122 res\_name ops.get\_op\_result\_name(self, other) 6124 if isinstance(other, Series) and not self.\_indexed\_same(other): - 6125 raise ValueError(Can only compare identically-labeled Series objects) 6127 lvalues self.\_values 6128 rvalues extract\_array(other, extract\_numpyTrue, extract\_rangeTrue) ValueError: Can only compare identically-labeled Series objects In [70]: pd.Series([foo, bar, baz]) pd.Series([foo]) --------------------------------------------------------------------------- ValueError Traceback (most recent call last) Cell In[70], line 1 ---- 1 pd.Series([foo, bar, baz]) pd.Series([foo]) File workpandaspandaspandascoreopscommon.py:76, in \_unpack\_zerodim\_and\_defer.locals.new\_method(self, other) 72 return NotImplemented 74 other item\_from\_zerodim(other) --- 76 return method(self, other) File workpandaspandaspandascorearraylike.py:40, in OpsMixin.\_\_eq\_\_(self, other) 38 unpack\_zerodim\_and\_defer(\_\_eq\_\_) 39 def \_\_eq\_\_(self, other): --- 40 return self.\_cmp\_method(other, operator.eq) File workpandaspandaspandascoreseries.py:6125, in Series.\_cmp\_method(self, other, op) 6122 res\_name ops.get\_op\_result\_name(self, other) 6124 if isinstance(other, Series) and not self.\_indexed\_same(other): - 6125 raise ValueError(Can only compare identically-labeled Series objects) 6127 lvalues self.\_values 6128 rvalues extract\_array(other, extract\_numpyTrue, extract\_rangeTrue) ValueError: Can only compare identically-labeled Series objects Combining overlapping data sets A problem occasionally arising is the combination of two similar data sets where values in one are preferred over the other. An example would be two data series representing a particular economic indicator where one is considered to be of higher quality. However, the lower quality series might extend further back in history or have more complete data coverage. As such, we would like to combine two DataFrame objects where missing values in one DataFrame are conditionally filled with like-labeled values from the other DataFrame. The function implementing this operation is combine\_first(), which we illustrate: In [71]: df1 pd.DataFrame( ....: {A: [1.0, np.nan, 3.0, 5.0, np.nan], B: [np.nan, 2.0, 3.0, np.nan, 6.0]} ....: ) ....: In [72]: df2 pd.DataFrame( ....: { ....: A: [5.0, 2.0, 4.0, np.nan, 3.0, 7.0], ....: B: [np.nan, np.nan, 3.0, 4.0, 6.0, 8.0], ....: } ....: ) ....: In [73]: df1 Out[73]: A B 0 1.0 NaN 1 NaN 2.0 2 3.0 3.0 3 5.0 NaN 4 NaN 6.0 In [74]: df2 Out[74]: A B 0 5.0 NaN 1 2.0 NaN 2 4.0 3.0 3 NaN 4.0 4 3.0 6.0 5 7.0 8.0 In [75]: df1.combine\_first(df2) Out[75]: A B 0 1.0 NaN 1 2.0 2.0 2 3.0 3.0 3 5.0 4.0 4 3.0 6.0 5 7.0 8.0 General DataFrame combine The combine\_first() method above calls the more general DataFrame.combine(). This method takes another DataFrame and a combiner function, aligns the input DataFrame and then passes the combiner function pairs of Series (i.e., columns whose names are the same). So, for instance, to reproduce combine\_first() as above: In [76]: def combiner(x, y): ....: return np.where(pd.isna(x), y, x) ....: In [77]: df1.combine(df2, combiner) Out[77]: A B 0 1.0 NaN 1 2.0 2.0 2 3.0 3.0 3 5.0 4.0 4 3.0 6.0 5 7.0 8.0 Descriptive statistics There exists a large number of methods for computing descriptive statistics and other related operations on Series, 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 (axis0, default), columns (axis1) For example: In [78]: df Out[78]: one two three a 1.394981 1.772517 NaN b 0.343054 1.912123 -0.050390 c 0.695246 1.478369 1.227435 d NaN 0.279344 -0.613172 In [79]: df.mean(0) Out[79]: one 0.811094 two 1.360588 three 0.187958 dtype: float64 In [80]: df.mean(1) Out[80]: a 1.583749 b 0.734929 c 1.133683 d -0.166914 dtype: float64 All such methods have a skipna option signaling whether to exclude missing data (True by default): In [81]: df.sum(0, skipnaFalse) Out[81]: one NaN two 5.442353 three NaN dtype: float64 In [82]: df.sum(axis1, skipnaTrue) Out[82]: a 3.167498 b 2.204786 c 3.401050 d -0.333828 dtype: float64 Combined with the broadcasting arithmetic behavior, one can describe various statistical procedures, like standardization (rendering data zero mean and standard deviation of 1), very concisely: In [83]: ts\_stand (df - df.mean()) df.std() In [84]: ts\_stand.std() Out[84]: one 1.0 two 1.0 three 1.0 dtype: float64 In [85]: xs\_stand df.sub(df.mean(1), axis0).div(df.std(1), axis0) In [86]: xs\_stand.std(1) Out[86]: a 1.0 b 1.0 c 1.0 d 1.0 dtype: float64 Note that methods like cumsum() and cumprod() preserve the location of NaN values. This is somewhat different from expanding() and rolling() since NaN behavior is furthermore dictated by a min\_periods parameter. In [87]: df.cumsum() Out[87]: one two three a 1.394981 1.772517 NaN b 1.738035 3.684640 -0.050390 c 2.433281 5.163008 1.177045 d NaN 5.442353 0.563873 Here is a quick reference summary table of common functions. Each also takes an optional level parameter which applies only if the object has a hierarchical index. Function Description count Number of non-NA observations sum Sum of values mean Mean of values median Arithmetic median of values min Minimum max Maximum mode Mode abs Absolute Value prod Product of values std Bessel-corrected sample standard deviation var Unbiased variance sem Standard error of the mean skew Sample skewness (3rd moment) kurt Sample kurtosis (4th moment) quantile Sample quantile (value at ) cumsum Cumulative sum cumprod Cumulative product cummax Cumulative maximum cummin Cumulative minimum Note that by chance some NumPy methods, like mean, std, and sum, will exclude NAs on Series input by default: In [88]: np.mean(df[one]) Out[88]: 0.8110935116651192 In [89]: np.mean(df[one].to\_numpy()) Out[89]: nan Series.nunique() will return the number of unique non-NA values in a Series: In [90]: series pd.Series(np.random.randn(500)) In [91]: series[20:500] np.nan In [92]: series[10:20] 5 In [93]: series.nunique() Out[93]: 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 (excluding NAs of course): In [94]: series pd.Series(np.random.randn(1000)) In [95]: series[::2] np.nan In [96]: series.describe() Out[96]: count 500.000000 mean -0.021292 std 1.015906 min -2.683763 25 -0.699070 50 -0.069718 75 0.714483 max 3.160915 dtype: float64 In [97]: frame pd.DataFrame(np.random.randn(1000, 5), columns[a, b, c, d, e]) In [98]: frame.iloc[::2] np.nan In [99]: frame.describe() Out[99]: a b c d e count 500.000000 500.000000 500.000000 500.000000 500.000000 mean 0.033387 0.030045 -0.043719 -0.051686 0.005979 std 1.017152 0.978743 1.025270 1.015988 1.006695 min -3.000951 -2.637901 -3.303099 -3.159200 -3.188821 25 -0.647623 -0.576449 -0.712369 -0.691338 -0.691115 50 0.047578 -0.021499 -0.023888 -0.032652 -0.025363 75 0.729907 0.775880 0.618896 0.670047 0.649748 max 2.740139 2.752332 3.004229 2.728702 3.240991 You can select specific percentiles to include in the output: In [100]: series.describe(percentiles[0.05, 0.25, 0.75, 0.95]) Out[100]: count 500.000000 mean -0.021292 std 1.015906 min -2.683763 5 -1.645423 25 -0.699070 50 -0.069718 75 0.714483 95 1.711409 max 3.160915 dtype: float64 By default, the median is always included. For a non-numerical Series object, describe() will give a simple summary of the number of unique values and most frequently occurring values: In [101]: s pd.Series([a, a, b, b, a, a, np.nan, c, d, a]) In [102]: s.describe() Out[102]: 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 [103]: frame pd.DataFrame({a: [Yes, Yes, No, No], b: range(4)}) In [104]: frame.describe() Out[104]: 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 behavior can be controlled by providing a list of types as includeexclude arguments. The special value all can also be used: In [105]: frame.describe(include[object]) Out[105]: a count 4 unique 2 top Yes freq 2 In [106]: frame.describe(include[number]) Out[106]: 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 [107]: frame.describe(includeall) Out[107]: 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 That feature relies on select\_dtypes. Refer to there for details about accepted inputs. Index of minmax values The idxmin() and idxmax() functions on Series and DataFrame compute the index labels with the minimum and maximum corresponding values: In [108]: s1 pd.Series(np.random.randn(5)) In [109]: s1 Out[109]: 0 1.118076 1 -0.352051 2 -1.242883 3 -1.277155 4 -0.641184 dtype: float64 In [110]: s1.idxmin(), s1.idxmax() Out[110]: (3, 0) In [111]: df1 pd.DataFrame(np.random.randn(5, 3), columns[A, B, C]) In [112]: df1 Out[112]: A B C 0 -0.327863 -0.946180 -0.137570 1 -0.186235 -0.257213 -0.486567 2 -0.507027 -0.871259 -0.111110 3 2.000339 -2.430505 0.089759 4 -0.321434 -0.033695 0.096271 In [113]: df1.idxmin(axis0) Out[113]: A 2 B 3 C 1 dtype: int64 In [114]: df1.idxmax(axis1) Out[114]: 0 C 1 A 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 [115]: df3 pd.DataFrame([2, 1, 1, 3, np.nan], columns[A], indexlist(edcba)) In [116]: df3 Out[116]: A e 2.0 d 1.0 c 1.0 b 3.0 a NaN In [117]: df3[A].idxmin() Out[117]: d Note idxmin and idxmax are called argmin and argmax in NumPy. Value counts (histogramming) mode The value\_counts() Series method computes a histogram of a 1D array of values. It can also be used as a function on regular arrays: In [118]: data np.random.randint(0, 7, size50) In [119]: data Out[119]: array([6, 6, 2, 3, 5, 3, 2, 5, 4, 5, 4, 3, 4, 5, 0, 2, 0, 4, 2, 0, 3, 2, 2, 5, 6, 5, 3, 4, 6, 4, 3, 5, 6, 4, 3, 6, 2, 6, 6, 2, 3, 4, 2, 1, 6, 2, 6, 1, 5, 4]) In [120]: s pd.Series(data) In [121]: s.value\_counts() Out[121]: 6 10 2 10 4 9 3 8 5 8 0 3 1 2 Name: count, dtype: int64 The value\_counts() method can be used to count combinations across multiple columns. By default all columns are used but a subset can be selected using the subset argument. In [122]: data {a: [1, 2, 3, 4], b: [x, x, y, y]} In [123]: frame pd.DataFrame(data) In [124]: frame.value\_counts() Out[124]: a b 1 x 1 2 x 1 3 y 1 4 y 1 Name: count, dtype: int64 Similarly, you can get the most frequently occurring value(s), i.e. the mode, of the values in a Series or DataFrame: In [125]: s5 pd.Series([1, 1, 3, 3, 3, 5, 5, 7, 7, 7]) In [126]: s5.mode() Out[126]: 0 3 1 7 dtype: int64 In [127]: df5 pd.DataFrame( .....: { .....: A: np.random.randint(0, 7, size50), .....: B: np.random.randint(-10, 15, size50), .....: } .....: ) .....: In [128]: df5.mode() Out[128]: A B 0 1.0 -9 1 NaN 10 2 NaN 13 Discretization and quantiling Continuous values can be discretized using the cut() (bins based on values) and qcut() (bins based on sample quantiles) functions: In [129]: arr np.random.randn(20) In [130]: factor pd.cut(arr, 4) In [131]: factor Out[131]: [(-0.251, 0.464], (-0.968, -0.251], (0.464, 1.179], (-0.251, 0.464], (-0.968, -0.251], ..., (-0.251, 0.464], (-0.968, -0.251], (-0.968, -0.251], (-0.968, -0.251], (-0.968, -0.251]] Length: 20 Categories (4, interval[float64, right]): [(-0.968, -0.251] (-0.251, 0.464] (0.464, 1.179] (1.179, 1.893]] In [132]: factor pd.cut(arr, [-5, -1, 0, 1, 5]) In [133]: factor Out[133]: [(0, 1], (-1, 0], (0, 1], (0, 1], (-1, 0], ..., (-1, 0], (-1, 0], (-1, 0], (-1, 0], (-1, 0]] Length: 20 Categories (4, interval[int64, right]): [(-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 [134]: arr np.random.randn(30) In [135]: factor pd.qcut(arr, [0, 0.25, 0.5, 0.75, 1]) In [136]: factor Out[136]: [(0.569, 1.184], (-2.278, -0.301], (-2.278, -0.301], (0.569, 1.184], (0.569, 1.184], ..., (-0.301, 0.569], (1.184, 2.346], (1.184, 2.346], (-0.301, 0.569], (-2.278, -0.301]] Length: 30 Categories (4, interval[float64, right]): [(-2.278, -0.301] (-0.301, 0.569] (0.569, 1.184] (1.184, 2.346]] We can also pass infinite values to define the bins: In [137]: arr np.random.randn(20) In [138]: factor pd.cut(arr, [-np.inf, 0, np.inf]) In [139]: factor Out[139]: [(-inf, 0.0], (0.0, inf], (0.0, inf], (-inf, 0.0], (-inf, 0.0], ..., (-inf, 0.0], (-inf, 0.0], (-inf, 0.0], (0.0, inf], (0.0, inf]] Length: 20 Categories (2, interval[float64, right]): [(-inf, 0.0] (0.0, inf]] Function application To apply your own or another librarys functions to pandas objects, you should be aware of the three methods below. The appropriate method to use depends on whether your function expects to operate on an entire DataFrame or Series, row- or column-wise, or elementwise. Tablewise Function Application: pipe() Row or Column-wise Function Application: apply() Aggregation API: agg() and transform() Applying Elementwise Functions: map() Tablewise function application DataFrames and Series can be passed into functions. However, if the function needs to be called in a chain, consider using the pipe() method. First some setup: In [140]: def extract\_city\_name(df): .....: .....: Chicago, IL - Chicago for city\_name column .....: .....: df[city\_name] df[city\_and\_code].str.split(,).str.get(0) .....: return df .....: In [141]: def add\_country\_name(df, country\_nameNone): .....: .....: Chicago - Chicago-US for city\_name column .....: .....: col city\_name .....: df[city\_and\_country] df[col] country\_name .....: return df .....: In [142]: df\_p pd.DataFrame({city\_and\_code: [Chicago, IL]}) extract\_city\_name and add\_country\_name are functions taking and returning DataFrames. Now compare the following: In [143]: add\_country\_name(extract\_city\_name(df\_p), country\_nameUS) Out[143]: city\_and\_code city\_name city\_and\_country 0 Chicago, IL Chicago ChicagoUS Is equivalent to: In [144]: df\_p.pipe(extract\_city\_name).pipe(add\_country\_name, country\_nameUS) Out[144]: city\_and\_code city\_name city\_and\_country 0 Chicago, IL Chicago ChicagoUS pandas encourages the second style, which is known as method chaining. pipe makes it easy to use your own or another librarys functions in method chains, alongside pandas methods. In the example above, the functions extract\_city\_name and add\_country\_name each expected a DataFrame as the first positional argument. What if the function you wish to apply takes its data as, say, the second argument? In this case, provide pipe with a tuple of (callable, data\_keyword). .pipe will route the DataFrame to the argument specified in the tuple. For example, we can fit a regression using statsmodels. Their API expects a formula first and a DataFrame as the second argument, data. We pass in the function, keyword pair (sm.ols, data) to pipe: In [147]: import statsmodels.formula.api as sm In [148]: bb pd.read\_csv(databaseball.csv, index\_colid) In [149]: ( .....: bb.query(h 0) .....: .assign(ln\_hlambda df: np.log(df.h)) .....: .pipe((sm.ols, data), hr ln\_h year g C(lg)) .....: .fit() .....: .summary() .....: ) .....: Out[149]: class statsmodels.iolib.summary.Summary OLS Regression Results Dep. Variable: hr R-squared: 0.685 Model: OLS Adj. R-squared: 0.665 Method: Least Squares F-statistic: 34.28 Date: Tue, 22 Nov 2022 Prob (F-statistic): 3.48e-15 Time: 05:34:17 Log-Likelihood: -205.92 No. Observations: 68 AIC: 421.8 Df Residuals: 63 BIC: 432.9 Df Model: 4 Covariance Type: nonrobust coef std err t Pt [0.025 0.975] ------------------------------------------------------------------------------- Intercept -8484.7720 4664.146 -1.819 0.074 -1.78e04 835.780 C(lg)[T.NL] -2.2736 1.325 -1.716 0.091 -4.922 0.375 ln\_h -1.3542 0.875 -1.547 0.127 -3.103 0.395 year 4.2277 2.324 1.819 0.074 -0.417 8.872 g 0.1841 0.029 6.258 0.000 0.125 0.243 Omnibus: 10.875 Durbin-Watson: 1.999 Prob(Omnibus): 0.004 Jarque-Bera (JB): 17.298 Skew: 0.537 Prob(JB): 0.000175 Kurtosis: 5.225 Cond. No. 1.49e07 Notes: [1] Standard Errors assume that the covariance matrix of the errors is correctly specified. [2] The condition number is large, 1.49e07. This might indicate that there are strong multicollinearity or other numerical problems. The pipe method is inspired by unix pipes and more recently dplyr and magrittr, which have introduced the popular () (read pipe) operator for R. The implementation of pipe here is quite clean and feels right at home in Python. We encourage you to view the source code of pipe(). 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, takes an optional axis argument: In [145]: df.apply(lambda x: np.mean(x)) Out[145]: one 0.811094 two 1.360588 three 0.187958 dtype: float64 In [146]: df.apply(lambda x: np.mean(x), axis1) Out[146]: a 1.583749 b 0.734929 c 1.133683 d -0.166914 dtype: float64 In [147]: df.apply(lambda x: x.max() - x.min()) Out[147]: one 1.051928 two 1.632779 three 1.840607 dtype: float64 In [148]: df.apply(np.cumsum) Out[148]: one two three a 1.394981 1.772517 NaN b 1.738035 3.684640 -0.050390 c 2.433281 5.163008 1.177045 d NaN 5.442353 0.563873 In [149]: df.apply(np.exp) Out[149]: one two three a 4.034899 5.885648 NaN b 1.409244 6.767440 0.950858 c 2.004201 4.385785 3.412466 d NaN 1.322262 0.541630 The apply() method will also dispatch on a string method name. In [150]: df.apply(mean) Out[150]: one 0.811094 two 1.360588 three 0.187958 dtype: float64 In [151]: df.apply(mean, axis1) Out[151]: a 1.583749 b 0.734929 c 1.133683 d -0.166914 dtype: float64 The return type of the function passed to apply() affects the type of the final output from DataFrame.apply for the default behaviour: If the applied function returns a Series, the final output is a DataFrame. The columns match the index of the Series returned by the applied function. If the applied function returns any other type, the final output is a Series. This default behaviour can be overridden using the result\_type, which accepts three options: reduce, broadcast, and expand. These will determine how list-likes return values expand (or not) to a DataFrame. 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 [152]: tsdf pd.DataFrame( .....: np.random.randn(1000, 3), .....: columns[A, B, C], .....: indexpd.date\_range(112000, periods1000), .....: ) .....: In [153]: tsdf.apply(lambda x: x.idxmax()) Out[153]: A 2000-08-06 B 2001-01-18 C 2001-07-18 dtype: datetime64[ns] You may also pass additional arguments and keyword arguments to the apply() method. In [154]: def subtract\_and\_divide(x, sub, divide1): .....: return (x - sub) divide .....: In [155]: df\_udf pd.DataFrame(np.ones((2, 2))) In [156]: df\_udf.apply(subtract\_and\_divide, args(5,), divide3) Out[156]: 0 1 0 -1.333333 -1.333333 1 -1.333333 -1.333333 Another useful feature is the ability to pass Series methods to carry out some Series operation on each column or row: In [157]: tsdf pd.DataFrame( .....: np.random.randn(10, 3), .....: columns[A, B, C], .....: indexpd.date\_range(112000, periods10), .....: ) .....: In [158]: tsdf.iloc[3:7] np.nan In [159]: tsdf Out[159]: A B C 2000-01-01 -0.158131 -0.232466 0.321604 2000-01-02 -1.810340 -3.105758 0.433834 2000-01-03 -1.209847 -1.156793 -0.136794 2000-01-04 NaN NaN NaN 2000-01-05 NaN NaN NaN 2000-01-06 NaN NaN NaN 2000-01-07 NaN NaN NaN 2000-01-08 -0.653602 0.178875 1.008298 2000-01-09 1.007996 0.462824 0.254472 2000-01-10 0.307473 0.600337 1.643950 In [160]: tsdf.apply(pd.Series.interpolate) Out[160]: A B C 2000-01-01 -0.158131 -0.232466 0.321604 2000-01-02 -1.810340 -3.105758 0.433834 2000-01-03 -1.209847 -1.156793 -0.136794 2000-01-04 -1.098598 -0.889659 0.092225 2000-01-05 -0.987349 -0.622526 0.321243 2000-01-06 -0.876100 -0.355392 0.550262 2000-01-07 -0.764851 -0.088259 0.779280 2000-01-08 -0.653602 0.178875 1.008298 2000-01-09 1.007996 0.462824 0.254472 2000-01-10 0.307473 0.600337 1.643950 Finally, apply() takes an argument raw which is False by default, which converts each row or column into a Series before applying the function. When set to True, the passed function will instead receive an ndarray object, which has positive performance implications if you do not need the indexing functionality. Aggregation API The aggregation API allows one to express possibly multiple aggregation operations in a single concise way. This API is similar across pandas objects, see groupby API, the window API, and the resample API. The entry point for aggregation is DataFrame.aggregate(), or the alias DataFrame.agg(). We will use a similar starting frame from above: In [161]: tsdf pd.DataFrame( .....: np.random.randn(10, 3), .....: columns[A, B, C], .....: indexpd.date\_range(112000, periods10), .....: ) .....: In [162]: tsdf.iloc[3:7] np.nan In [163]: tsdf Out[163]: A B C 2000-01-01 1.257606 1.004194 0.167574 2000-01-02 -0.749892 0.288112 -0.757304 2000-01-03 -0.207550 -0.298599 0.116018 2000-01-04 NaN NaN NaN 2000-01-05 NaN NaN NaN 2000-01-06 NaN NaN NaN 2000-01-07 NaN NaN NaN 2000-01-08 0.814347 -0.257623 0.869226 2000-01-09 -0.250663 -1.206601 0.896839 2000-01-10 2.169758 -1.333363 0.283157 Using a single function is equivalent to apply(). You can also pass named methods as strings. These will return a Series of the aggregated output: In [164]: tsdf.agg(lambda x: np.sum(x)) Out[164]: A 3.033606 B -1.803879 C 1.575510 dtype: float64 In [165]: tsdf.agg(sum) Out[165]: A 3.033606 B -1.803879 C 1.575510 dtype: float64 these are equivalent to a .sum() because we are aggregating on a single function In [166]: tsdf.sum() Out[166]: A 3.033606 B -1.803879 C 1.575510 dtype: float64 Single aggregations on a Series this will return a scalar value: In [167]: tsdf[A].agg(sum) Out[167]: 3.033606102414146 Aggregating with multiple functions You can pass multiple aggregation arguments as a list. The results of each of the passed functions will be a row in the resulting DataFrame. These are naturally named from the aggregation function. In [168]: tsdf.agg([sum]) Out[168]: A B C sum 3.033606 -1.803879 1.57551 Multiple functions yield multiple rows: In [169]: tsdf.agg([sum, mean]) Out[169]: A B C sum 3.033606 -1.803879 1.575510 mean 0.505601 -0.300647 0.262585 On a Series, multiple functions return a Series, indexed by the function names: In [170]: tsdf[A].agg([sum, mean]) Out[170]: sum 3.033606 mean 0.505601 Name: A, dtype: float64 Passing a lambda function will yield a lambda named row: In [171]: tsdf[A].agg([sum, lambda x: x.mean()]) Out[171]: sum 3.033606 lambda 0.505601 Name: A, dtype: float64 Passing a named function will yield that name for the row: In [172]: def mymean(x): .....: return x.mean() .....: In [173]: tsdf[A].agg([sum, mymean]) Out[173]: sum 3.033606 mymean 0.505601 Name: A, dtype: float64 Aggregating with a dict Passing a dictionary of column names to a scalar or a list of scalars, to DataFrame.agg allows you to customize which functions are applied to which columns. Note that the results are not in any particular order, you can use an OrderedDict instead to guarantee ordering. In [174]: tsdf.agg({A: mean, B: sum}) Out[174]: A 0.505601 B -1.803879 dtype: float64 Passing a list-like will generate a DataFrame output. You will get a matrix-like output of all of the aggregators. The output will consist of all unique functions. Those that are not noted for a particular column will be NaN: In [175]: tsdf.agg({A: [mean, min], B: sum}) Out[175]: A B mean 0.505601 NaN min -0.749892 NaN sum NaN -1.803879 Custom describe With .agg() it is possible to easily create a custom describe function, similar to the built in describe function. In [176]: from functools import partial In [177]: q\_25 partial(pd.Series.quantile, q0.25) In [178]: q\_25.\_\_name\_\_ 25 In [179]: q\_75 partial(pd.Series.quantile, q0.75) In [180]: q\_75.\_\_name\_\_ 75 In [181]: tsdf.agg([count, mean, std, min, q\_25, median, q\_75, max]) Out[181]: A B C count 6.000000 6.000000 6.000000 mean 0.505601 -0.300647 0.262585 std 1.103362 0.887508 0.606860 min -0.749892 -1.333363 -0.757304 25 -0.239885 -0.979600 0.128907 median 0.303398 -0.278111 0.225365 75 1.146791 0.151678 0.722709 max 2.169758 1.004194 0.896839 Transform API The transform() method returns an object that is indexed the same (same size) as the original. This API allows you to provide multiple operations at the same time rather than one-by-one. Its API is quite similar to the .agg API. We create a frame similar to the one used in the above sections. In [182]: tsdf pd.DataFrame( .....: np.random.randn(10, 3), .....: columns[A, B, C], .....: indexpd.date\_range(112000, periods10), .....: ) .....: In [183]: tsdf.iloc[3:7] np.nan In [184]: tsdf Out[184]: A B C 2000-01-01 -0.428759 -0.864890 -0.675341 2000-01-02 -0.168731 1.338144 -1.279321 2000-01-03 -1.621034 0.438107 0.903794 2000-01-04 NaN NaN NaN 2000-01-05 NaN NaN NaN 2000-01-06 NaN NaN NaN 2000-01-07 NaN NaN NaN 2000-01-08 0.254374 -1.240447 -0.201052 2000-01-09 -0.157795 0.791197 -1.144209 2000-01-10 -0.030876 0.371900 0.061932 Transform the entire frame. .transform() allows input functions as: a NumPy function, a string function name or a user defined function. In [185]: tsdf.transform(np.abs) Out[185]: A B C 2000-01-01 0.428759 0.864890 0.675341 2000-01-02 0.168731 1.338144 1.279321 2000-01-03 1.621034 0.438107 0.903794 2000-01-04 NaN NaN NaN 2000-01-05 NaN NaN NaN 2000-01-06 NaN NaN NaN 2000-01-07 NaN NaN NaN 2000-01-08 0.254374 1.240447 0.201052 2000-01-09 0.157795 0.791197 1.144209 2000-01-10 0.030876 0.371900 0.061932 In [186]: tsdf.transform(abs) Out[186]: A B C 2000-01-01 0.428759 0.864890 0.675341 2000-01-02 0.168731 1.338144 1.279321 2000-01-03 1.621034 0.438107 0.903794 2000-01-04 NaN NaN NaN 2000-01-05 NaN NaN NaN 2000-01-06 NaN NaN NaN 2000-01-07 NaN NaN NaN 2000-01-08 0.254374 1.240447 0.201052 2000-01-09 0.157795 0.791197 1.144209 2000-01-10 0.030876 0.371900 0.061932 In [187]: tsdf.transform(lambda x: x.abs()) Out[187]: A B C 2000-01-01 0.428759 0.864890 0.675341 2000-01-02 0.168731 1.338144 1.279321 2000-01-03 1.621034 0.438107 0.903794 2000-01-04 NaN NaN NaN 2000-01-05 NaN NaN NaN 2000-01-06 NaN NaN NaN 2000-01-07 NaN NaN NaN 2000-01-08 0.254374 1.240447 0.201052 2000-01-09 0.157795 0.791197 1.144209 2000-01-10 0.030876 0.371900 0.061932 Here transform() received a single function; this is equivalent to a ufunc application. In [188]: np.abs(tsdf) Out[188]: A B C 2000-01-01 0.428759 0.864890 0.675341 2000-01-02 0.168731 1.338144 1.279321 2000-01-03 1.621034 0.438107 0.903794 2000-01-04 NaN NaN NaN 2000-01-05 NaN NaN NaN 2000-01-06 NaN NaN NaN 2000-01-07 NaN NaN NaN 2000-01-08 0.254374 1.240447 0.201052 2000-01-09 0.157795 0.791197 1.144209 2000-01-10 0.030876 0.371900 0.061932 Passing a single function to .transform() with a Series will yield a single Series in return. In [189]: tsdf[A].transform(np.abs) Out[189]: 2000-01-01 0.428759 2000-01-02 0.168731 2000-01-03 1.621034 2000-01-04 NaN 2000-01-05 NaN 2000-01-06 NaN 2000-01-07 NaN 2000-01-08 0.254374 2000-01-09 0.157795 2000-01-10 0.030876 Freq: D, Name: A, dtype: float64 Transform with multiple functions Passing multiple functions will yield a column MultiIndexed DataFrame. The first level will be the original frame column names; the second level will be the names of the transforming functions. In [190]: tsdf.transform([np.abs, lambda x: x 1]) Out[190]: A B C absolute lambda absolute lambda absolute lambda 2000-01-01 0.428759 0.571241 0.864890 0.135110 0.675341 0.324659 2000-01-02 0.168731 0.831269 1.338144 2.338144 1.279321 -0.279321 2000-01-03 1.621034 -0.621034 0.438107 1.438107 0.903794 1.903794 2000-01-04 NaN NaN NaN NaN NaN NaN 2000-01-05 NaN NaN NaN NaN NaN NaN 2000-01-06 NaN NaN NaN NaN NaN NaN 2000-01-07 NaN NaN NaN NaN NaN NaN 2000-01-08 0.254374 1.254374 1.240447 -0.240447 0.201052 0.798948 2000-01-09 0.157795 0.842205 0.791197 1.791197 1.144209 -0.144209 2000-01-10 0.030876 0.969124 0.371900 1.371900 0.061932 1.061932 Passing multiple functions to a Series will yield a DataFrame. The resulting column names will be the transforming functions. In [191]: tsdf[A].transform([np.abs, lambda x: x 1]) Out[191]: absolute lambda 2000-01-01 0.428759 0.571241 2000-01-02 0.168731 0.831269 2000-01-03 1.621034 -0.621034 2000-01-04 NaN NaN 2000-01-05 NaN NaN 2000-01-06 NaN NaN 2000-01-07 NaN NaN 2000-01-08 0.254374 1.254374 2000-01-09 0.157795 0.842205 2000-01-10 0.030876 0.969124 Transforming with a dict Passing a dict of functions will allow selective transforming per column. In [192]: tsdf.transform({A: np.abs, B: lambda x: x 1}) Out[192]: A B 2000-01-01 0.428759 0.135110 2000-01-02 0.168731 2.338144 2000-01-03 1.621034 1.438107 2000-01-04 NaN NaN 2000-01-05 NaN NaN 2000-01-06 NaN NaN 2000-01-07 NaN NaN 2000-01-08 0.254374 -0.240447 2000-01-09 0.157795 1.791197 2000-01-10 0.030876 1.371900 Passing a dict of lists will generate a MultiIndexed DataFrame with these selective transforms. In [193]: tsdf.transform({A: np.abs, B: [lambda x: x 1, sqrt]}) Out[193]: A B absolute lambda sqrt 2000-01-01 0.428759 0.135110 NaN 2000-01-02 0.168731 2.338144 1.156782 2000-01-03 1.621034 1.438107 0.661897 2000-01-04 NaN NaN NaN 2000-01-05 NaN NaN NaN 2000-01-06 NaN NaN NaN 2000-01-07 NaN NaN NaN 2000-01-08 0.254374 -0.240447 NaN 2000-01-09 0.157795 1.791197 0.889493 2000-01-10 0.030876 1.371900 0.609836 Applying elementwise functions Since not all functions can be vectorized (accept NumPy arrays and return another array or value), the methods map() on DataFrame and analogously map() on Series accept any Python function taking a single value and returning a single value. For example: In [194]: df4 df.copy() In [195]: df4 Out[195]: one two three a 1.394981 1.772517 NaN b 0.343054 1.912123 -0.050390 c 0.695246 1.478369 1.227435 d NaN 0.279344 -0.613172 In [196]: def f(x): .....: return len(str(x)) .....: In [197]: df4[one].map(f) Out[197]: a 18 b 19 c 18 d 3 Name: one, dtype: int64 In [198]: df4.map(f) Out[198]: one two three a 18 17 3 b 19 18 20 c 18 18 16 d 3 19 19 Series.map() has an additional feature; it can be used to easily link or map values defined by a secondary series. This is closely related to mergingjoining functionality: In [199]: s pd.Series( .....: [six, seven, six, seven, six], index[a, b, c, d, e] .....: ) .....: In [200]: t pd.Series({six: 6.0, seven: 7.0}) In [201]: s Out[201]: a six b seven c six d seven e six dtype: object In [202]: s.map(t) Out[202]: a 6.0 b 7.0 c 6.0 d 7.0 e 6.0 dtype: float64 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 [203]: s pd.Series(np.random.randn(5), index[a, b, c, d, e]) In [204]: s Out[204]: a 1.695148 b 1.328614 c 1.234686 d -0.385845 e -1.326508 dtype: float64 In [205]: s.reindex([e, b, f, d]) Out[205]: e -1.326508 b 1.328614 f NaN d -0.385845 dtype: float64 Here, the f label was not contained in the Series and hence appears as NaN in the result. With a DataFrame, you can simultaneously reindex the index and columns: In [206]: df Out[206]: one two three a 1.394981 1.772517 NaN b 0.343054 1.912123 -0.050390 c 0.695246 1.478369 1.227435 d NaN 0.279344 -0.613172 In [207]: df.reindex(index[c, f, b], columns[three, two, one]) Out[207]: three two one c 1.227435 1.478369 0.695246 f NaN NaN NaN b -0.050390 1.912123 0.343054 Note that the Index objects containing the actual axis labels can be shared between objects. So if we have a Series and a DataFrame, the following can be done: In [208]: rs s.reindex(df.index) In [209]: rs Out[209]: a 1.695148 b 1.328614 c 1.234686 d -0.385845 dtype: float64 In [210]: rs.index is df.index Out[210]: True This means that the reindexed Seriess index is the same Python object as the DataFrames index. DataFrame.reindex() also supports an axis-style calling convention, where you specify a single labels argument and the axis it applies to. In [211]: df.reindex([c, f, b], axisindex) Out[211]: one two three c 0.695246 1.478369 1.227435 f NaN NaN NaN b 0.343054 1.912123 -0.050390 In [212]: df.reindex([three, two, one], axiscolumns) Out[212]: three two one a NaN 1.772517 1.394981 b -0.050390 1.912123 0.343054 c 1.227435 1.478369 0.695246 d -0.613172 0.279344 NaN See also MultiIndex Advanced Indexing is an even more concise way of doing reindexing. Note When writing performance-sensitive code, there is a good reason to spend some time becoming a reindexing ninja: many operations are faster on pre-aligned data. Adding two unaligned DataFrames internally triggers a reindexing step. For exploratory analysis you will hardly notice the difference (because reindex has been heavily optimized), but when CPU cycles matter sprinkling a few explicit reindex calls here and there can have an impact. 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. While the syntax for this is straightforward albeit verbose, it is a common enough operation that the reindex\_like() method is available to make this simpler: In [213]: df2 df.reindex([a, b, c], columns[one, two]) In [214]: df3 df2 - df2.mean() In [215]: df2 Out[215]: one two a 1.394981 1.772517 b 0.343054 1.912123 c 0.695246 1.478369 In [216]: df3 Out[216]: one two a 0.583888 0.051514 b -0.468040 0.191120 c -0.115848 -0.242634 In [217]: df.reindex\_like(df2) Out[217]: one two a 1.394981 1.772517 b 0.343054 1.912123 c 0.695246 1.478369 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): joinouter: take the union of the indexes (default) joinleft: use the calling objects index joinright: use the passed objects index joininner: intersect the indexes It returns a tuple with both of the reindexed Series: In [218]: s pd.Series(np.random.randn(5), index[a, b, c, d, e]) In [219]: s1 s[:4] In [220]: s2 s[1:] In [221]: s1.align(s2) Out[221]: (a -0.186646 b -1.692424 c -0.303893 d -1.425662 e NaN dtype: float64, a NaN b -1.692424 c -0.303893 d -1.425662 e 1.114285 dtype: float64) In [222]: s1.align(s2, joininner) Out[222]: (b -1.692424 c -0.303893 d -1.425662 dtype: float64, b -1.692424 c -0.303893 d -1.425662 dtype: float64) In [223]: s1.align(s2, joinleft) Out[223]: (a -0.186646 b -1.692424 c -0.303893 d -1.425662 dtype: float64, a NaN b -1.692424 c -0.303893 d -1.425662 dtype: float64) For DataFrames, the join method will be applied to both the index and the columns by default: In [224]: df.align(df2, joininner) Out[224]: ( one two a 1.394981 1.772517 b 0.343054 1.912123 c 0.695246 1.478369, one two a 1.394981 1.772517 b 0.343054 1.912123 c 0.695246 1.478369) You can also pass an axis option to only align on the specified axis: In [225]: df.align(df2, joininner, axis0) Out[225]: ( one two three a 1.394981 1.772517 NaN b 0.343054 1.912123 -0.050390 c 0.695246 1.478369 1.227435, one two a 1.394981 1.772517 b 0.343054 1.912123 c 0.695246 1.478369) If you pass a Series to DataFrame.align(), you can choose to align both objects either on the DataFrames index or columns using the axis argument: In [226]: df.align(df2.iloc[0], axis1) Out[226]: ( one three two a 1.394981 NaN 1.772517 b 0.343054 -0.050390 1.912123 c 0.695246 1.227435 1.478369 d NaN -0.613172 0.279344, one 1.394981 three NaN two 1.772517 Name: a, dtype: float64) Filling while reindexing reindex() takes an optional parameter method which is a filling method chosen from the following table: Method Action pad ffill Fill values forward bfill backfill Fill values backward nearest Fill from the nearest index value We illustrate these fill methods on a simple Series: In [227]: rng pd.date\_range(132000, periods8) In [228]: ts pd.Series(np.random.randn(8), indexrng) In [229]: ts2 ts.iloc[[0, 3, 6]] In [230]: ts Out[230]: 2000-01-03 0.183051 2000-01-04 0.400528 2000-01-05 -0.015083 2000-01-06 2.395489 2000-01-07 1.414806 2000-01-08 0.118428 2000-01-09 0.733639 2000-01-10 -0.936077 Freq: D, dtype: float64 In [231]: ts2 Out[231]: 2000-01-03 0.183051 2000-01-06 2.395489 2000-01-09 0.733639 Freq: 3D, dtype: float64 In [232]: ts2.reindex(ts.index) Out[232]: 2000-01-03 0.183051 2000-01-04 NaN 2000-01-05 NaN 2000-01-06 2.395489 2000-01-07 NaN 2000-01-08 NaN 2000-01-09 0.733639 2000-01-10 NaN Freq: D, dtype: float64 In [233]: ts2.reindex(ts.index, methodffill) Out[233]: 2000-01-03 0.183051 2000-01-04 0.183051 2000-01-05 0.183051 2000-01-06 2.395489 2000-01-07 2.395489 2000-01-08 2.395489 2000-01-09 0.733639 2000-01-10 0.733639 Freq: D, dtype: float64 In [234]: ts2.reindex(ts.index, methodbfill) Out[234]: 2000-01-03 0.183051 2000-01-04 2.395489 2000-01-05 2.395489 2000-01-06 2.395489 2000-01-07 0.733639 2000-01-08 0.733639 2000-01-09 0.733639 2000-01-10 NaN Freq: D, dtype: float64 In [235]: ts2.reindex(ts.index, methodnearest) Out[235]: 2000-01-03 0.183051 2000-01-04 0.183051 2000-01-05 2.395489 2000-01-06 2.395489 2000-01-07 2.395489 2000-01-08 0.733639 2000-01-09 0.733639 2000-01-10 0.733639 Freq: D, dtype: float64 These methods require that the indexes are ordered increasing or decreasing. Note that the same result could have been achieved using ffill (except for methodnearest) or interpolate: In [236]: ts2.reindex(ts.index).ffill() Out[236]: 2000-01-03 0.183051 2000-01-04 0.183051 2000-01-05 0.183051 2000-01-06 2.395489 2000-01-07 2.395489 2000-01-08 2.395489 2000-01-09 0.733639 2000-01-10 0.733639 Freq: D, dtype: float64 reindex() will raise a ValueError if the index is not monotonically increasing or decreasing. fillna() and interpolate() will not perform any checks on the order of the index. Limits on filling while reindexing The limit and tolerance arguments provide additional control over filling while reindexing. Limit specifies the maximum count of consecutive matches: In [237]: ts2.reindex(ts.index, methodffill, limit1) Out[237]: 2000-01-03 0.183051 2000-01-04 0.183051 2000-01-05 NaN 2000-01-06 2.395489 2000-01-07 2.395489 2000-01-08 NaN 2000-01-09 0.733639 2000-01-10 0.733639 Freq: D, dtype: float64 In contrast, tolerance specifies the maximum distance between the index and indexer values: In [238]: ts2.reindex(ts.index, methodffill, tolerance1 day) Out[238]: 2000-01-03 0.183051 2000-01-04 0.183051 2000-01-05 NaN 2000-01-06 2.395489 2000-01-07 2.395489 2000-01-08 NaN 2000-01-09 0.733639 2000-01-10 0.733639 Freq: D, dtype: float64 Notice that when used on a DatetimeIndex, TimedeltaIndex or PeriodIndex, tolerance will coerced into a Timedelta if possible. This allows you to specify tolerance with appropriate strings. 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 [239]: df Out[239]: one two three a 1.394981 1.772517 NaN b 0.343054 1.912123 -0.050390 c 0.695246 1.478369 1.227435 d NaN 0.279344 -0.613172 In [240]: df.drop([a, d], axis0) Out[240]: one two three b 0.343054 1.912123 -0.050390 c 0.695246 1.478369 1.227435 In [241]: df.drop([one], axis1) Out[241]: two three a 1.772517 NaN b 1.912123 -0.050390 c 1.478369 1.227435 d 0.279344 -0.613172 Note that the following also works, but is a bit less obvious clean: In [242]: df.reindex(df.index.difference([a, d])) Out[242]: one two three b 0.343054 1.912123 -0.050390 c 0.695246 1.478369 1.227435 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 [243]: s Out[243]: a -0.186646 b -1.692424 c -0.303893 d -1.425662 e 1.114285 dtype: float64 In [244]: s.rename(str.upper) Out[244]: A -0.186646 B -1.692424 C -0.303893 D -1.425662 E 1.114285 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). A dict or Series can also be used: In [245]: df.rename( .....: columns{one: foo, two: bar}, .....: index{a: apple, b: banana, d: durian}, .....: ) .....: Out[245]: foo bar three apple 1.394981 1.772517 NaN banana 0.343054 1.912123 -0.050390 c 0.695246 1.478369 1.227435 durian NaN 0.279344 -0.613172 If the mapping doesnt include a columnindex label, it isnt renamed. Note that extra labels in the mapping dont throw an error. DataFrame.rename() also supports an axis-style calling convention, where you specify a single mapper and the axis to apply that mapping to. In [246]: df.rename({one: foo, two: bar}, axiscolumns) Out[246]: foo bar three a 1.394981 1.772517 NaN b 0.343054 1.912123 -0.050390 c 0.695246 1.478369 1.227435 d NaN 0.279344 -0.613172 In [247]: df.rename({a: apple, b: banana, d: durian}, axisindex) Out[247]: one two three apple 1.394981 1.772517 NaN banana 0.343054 1.912123 -0.050390 c 0.695246 1.478369 1.227435 durian NaN 0.279344 -0.613172 Finally, rename() also accepts a scalar or list-like for altering the Series.name attribute. In [248]: s.rename(scalar-name) Out[248]: a -0.186646 b -1.692424 c -0.303893 d -1.425662 e 1.114285 Name: scalar-name, dtype: float64 The methods DataFrame.rename\_axis() and Series.rename\_axis() allow specific names of a MultiIndex to be changed (as opposed to the labels). In [249]: df pd.DataFrame( .....: {x: [1, 2, 3, 4, 5, 6], y: [10, 20, 30, 40, 50, 60]}, .....: indexpd.MultiIndex.from\_product( .....: [[a, b, c], [1, 2]], names[let, num] .....: ), .....: ) .....: In [250]: df Out[250]: x y let num a 1 1 10 2 2 20 b 1 3 30 2 4 40 c 1 5 50 2 6 60 In [251]: df.rename\_axis(index{let: abc}) Out[251]: x y abc num a 1 1 10 2 2 20 b 1 3 30 2 4 40 c 1 5 50 2 6 60 In [252]: df.rename\_axis(indexstr.upper) Out[252]: x y LET NUM a 1 1 10 2 2 20 b 1 3 30 2 4 40 c 1 5 50 2 6 60 Iteration The behavior of basic iteration over pandas objects depends on the type. When iterating over a Series, it is regarded as array-like, and basic iteration produces the values. DataFrames follow the dict-like convention of iterating over the keys of the objects. In short, basic iteration (for i in object) produces: Series: values DataFrame: column labels Thus, for example, iterating over a DataFrame gives you the column names: In [253]: df pd.DataFrame( .....: {col1: np.random.randn(3), col2: np.random.randn(3)}, index[a, b, c] .....: ) .....: In [254]: for col in df: .....: print(col) .....: col1 col2 pandas objects also have the dict-like items() method to iterate over the (key, value) pairs. To iterate over the rows of a DataFrame, you can use the following methods: iterrows(): Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications. itertuples(): Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than iterrows(), and is in most cases preferable to use to iterate over the values of a DataFrame. Warning Iterating through pandas objects is generally slow. In many cases, iterating manually over the rows is not needed and can be avoided with one of the following approaches: Look for a vectorized solution: many operations can be performed using built-in methods or NumPy functions, (boolean) indexing, When you have a function that cannot work on the full DataFrameSeries at once, it is better to use apply() instead of iterating over the values. See the docs on function application. If you need to do iterative manipulations on the values but performance is important, consider writing the inner loop with cython or numba. See the enhancing performance section for some examples of this approach. Warning You should never modify something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect! For example, in the following case setting the value has no effect: In [255]: df pd.DataFrame({a: [1, 2, 3], b: [a, b, c]}) In [256]: for index, row in df.iterrows(): .....: row[a] 10 .....: In [257]: df Out[257]: a b 0 1 a 1 2 b 2 3 c items Consistent with the dict-like interface, items() iterates through key-value pairs: Series: (index, scalar value) pairs DataFrame: (column, Series) pairs For example: In [258]: for label, ser in df.items(): .....: print(label) .....: print(ser) .....: a 0 1 1 2 2 3 Name: a, dtype: int64 b 0 a 1 b 2 c Name: b, dtype: object iterrows iterrows() allows you to iterate through the rows of a DataFrame as Series objects. It returns an iterator yielding each index value along with a Series containing the data in each row: In [259]: for row\_index, row in df.iterrows(): .....: print(row\_index, row, sepn) .....: 0 a 1 b a Name: 0, dtype: object 1 a 2 b b Name: 1, dtype: object 2 a 3 b c Name: 2, dtype: object Note Because iterrows() returns a Series for each row, it does not preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example, In [260]: df\_orig pd.DataFrame([[1, 1.5]], columns[int, float]) In [261]: df\_orig.dtypes Out[261]: int int64 float float64 dtype: object In [262]: row next(df\_orig.iterrows())[1] In [263]: row Out[263]: int 1.0 float 1.5 Name: 0, dtype: float64 All values in row, returned as a Series, are now upcasted to floats, also the original integer value in column x: In [264]: row[int].dtype Out[264]: dtype(float64) In [265]: df\_orig[int].dtype Out[265]: dtype(int64) To preserve dtypes while iterating over the rows, it is better to use itertuples() which returns namedtuples of the values and which is generally much faster than iterrows(). For instance, a contrived way to transpose the DataFrame would be: In [266]: df2 pd.DataFrame({x: [1, 2, 3], y: [4, 5, 6]}) In [267]: print(df2) x y 0 1 4 1 2 5 2 3 6 In [268]: print(df2.T) 0 1 2 x 1 2 3 y 4 5 6 In [269]: df2\_t pd.DataFrame({idx: values for idx, values in df2.iterrows()}) In [270]: print(df2\_t) 0 1 2 x 1 2 3 y 4 5 6 itertuples The itertuples() method will return an iterator yielding a namedtuple for each row in the DataFrame. The first element of the tuple will be the rows corresponding index value, while the remaining values are the row values. For instance: In [271]: for row in df.itertuples(): .....: print(row) .....: Pandas(Index0, a1, ba) Pandas(Index1, a2, bb) Pandas(Index2, a3, bc) This method does not convert the row to a Series object; it merely returns the values inside a namedtuple. Therefore, itertuples() preserves the data type of the values and is generally faster as iterrows(). Note The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (255), regular tuples are returned. .dt accessor Series has an accessor to succinctly return datetime like properties for the values of the Series, if it is a datetimeperiod like Series. This will return a Series, indexed like the existing Series. datetime In [272]: s pd.Series(pd.date\_range(20130101 09:10:12, periods4)) In [273]: s Out[273]: 0 2013-01-01 09:10:12 1 2013-01-02 09:10:12 2 2013-01-03 09:10:12 3 2013-01-04 09:10:12 dtype: datetime64[ns] In [274]: s.dt.hour Out[274]: 0 9 1 9 2 9 3 9 dtype: int32 In [275]: s.dt.second Out[275]: 0 12 1 12 2 12 3 12 dtype: int32 In [276]: s.dt.day Out[276]: 0 1 1 2 2 3 3 4 dtype: int32 This enables nice expressions like this: In [277]: s[s.dt.day 2] Out[277]: 1 2013-01-02 09:10:12 dtype: datetime64[ns] You can easily produces tz aware transformations: In [278]: stz s.dt.tz\_localize(USEastern) In [279]: stz Out[279]: 0 2013-01-01 09:10:12-05:00 1 2013-01-02 09:10:12-05:00 2 2013-01-03 09:10:12-05:00 3 2013-01-04 09:10:12-05:00 dtype: datetime64[ns, USEastern] In [280]: stz.dt.tz Out[280]: DstTzInfo USEastern LMT-1 day, 19:04:00 STD You can also chain these types of operations: In [281]: s.dt.tz\_localize(UTC).dt.tz\_convert(USEastern) Out[281]: 0 2013-01-01 04:10:12-05:00 1 2013-01-02 04:10:12-05:00 2 2013-01-03 04:10:12-05:00 3 2013-01-04 04:10:12-05:00 dtype: datetime64[ns, USEastern] You can also format datetime values as strings with Series.dt.strftime() which supports the same format as the standard strftime(). DatetimeIndex In [282]: s pd.Series(pd.date\_range(20130101, periods4)) In [283]: s Out[283]: 0 2013-01-01 1 2013-01-02 2 2013-01-03 3 2013-01-04 dtype: datetime64[ns] In [284]: s.dt.strftime(Ymd) Out[284]: 0 20130101 1 20130102 2 20130103 3 20130104 dtype: object PeriodIndex In [285]: s pd.Series(pd.period\_range(20130101, periods4)) In [286]: s Out[286]: 0 2013-01-01 1 2013-01-02 2 2013-01-03 3 2013-01-04 dtype: period[D] In [287]: s.dt.strftime(Ymd) Out[287]: 0 20130101 1 20130102 2 20130103 3 20130104 dtype: object The .dt accessor works for period and timedelta dtypes. period In [288]: s pd.Series(pd.period\_range(20130101, periods4, freqD)) In [289]: s Out[289]: 0 2013-01-01 1 2013-01-02 2 2013-01-03 3 2013-01-04 dtype: period[D] In [290]: s.dt.year Out[290]: 0 2013 1 2013 2 2013 3 2013 dtype: int64 In [291]: s.dt.day Out[291]: 0 1 1 2 2 3 3 4 dtype: int64 timedelta In [292]: s pd.Series(pd.timedelta\_range(1 day 00:00:05, periods4, freqs)) In [293]: s Out[293]: 0 1 days 00:00:05 1 1 days 00:00:06 2 1 days 00:00:07 3 1 days 00:00:08 dtype: timedelta64[ns] In [294]: s.dt.days Out[294]: 0 1 1 1 2 1 3 1 dtype: int64 In [295]: s.dt.seconds Out[295]: 0 5 1 6 2 7 3 8 dtype: int32 In [296]: s.dt.components Out[296]: days hours minutes seconds milliseconds microseconds nanoseconds 0 1 0 0 5 0 0 0 1 1 0 0 6 0 0 0 2 1 0 0 7 0 0 0 3 1 0 0 8 0 0 0 Note Series.dt will raise a TypeError if you access with a non-datetime-like values. Vectorized string methods Series is equipped with a set of string processing methods that make it easy to operate on each element of the array. Perhaps most importantly, these methods exclude missingNA values automatically. These are accessed via the Seriess str attribute and generally have names matching the equivalent (scalar) built-in string methods. For example: In [297]: s pd.Series( .....: [A, B, C, Aaba, Baca, np.nan, CABA, dog, cat], dtypestring .....: ) .....: In [298]: s.str.lower() Out[298]: 0 a 1 b 2 c 3 aaba 4 baca 5 NA 6 caba 7 dog 8 cat dtype: string Powerful pattern-matching methods are provided as well, but note that pattern-matching generally uses regular expressions by default (and in some cases always uses them). Note Prior to pandas 1.0, string methods were only available on object -dtype Series. pandas 1.0 added the StringDtype which is dedicated to strings. See Text data types for more. Please see Vectorized String Methods for a complete description. Sorting pandas supports three kinds of sorting: sorting by index labels, sorting by column values, and sorting by a combination of both. By index The Series.sort\_index() and DataFrame.sort\_index() methods are used to sort a pandas object by its index levels. In [299]: 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 [300]: unsorted\_df df.reindex( .....: index[a, d, c, b], columns[three, two, one] .....: ) .....: In [301]: unsorted\_df Out[301]: three two one a NaN -1.152244 0.562973 d -0.252916 -0.109597 NaN c 1.273388 -0.167123 0.640382 b -0.098217 0.009797 -1.299504 DataFrame In [302]: unsorted\_df.sort\_index() Out[302]: three two one a NaN -1.152244 0.562973 b -0.098217 0.009797 -1.299504 c 1.273388 -0.167123 0.640382 d -0.252916 -0.109597 NaN In [303]: unsorted\_df.sort\_index(ascendingFalse) Out[303]: three two one d -0.252916 -0.109597 NaN c 1.273388 -0.167123 0.640382 b -0.098217 0.009797 -1.299504 a NaN -1.152244 0.562973 In [304]: unsorted\_df.sort\_index(axis1) Out[304]: one three two a 0.562973 NaN -1.152244 d NaN -0.252916 -0.109597 c 0.640382 1.273388 -0.167123 b -1.299504 -0.098217 0.009797 Series In [305]: unsorted\_df[three].sort\_index() Out[305]: a NaN b -0.098217 c 1.273388 d -0.252916 Name: three, dtype: float64 Sorting by index also supports a key parameter that takes a callable function to apply to the index being sorted. For MultiIndex objects, the key is applied per-level to the levels specified by level. In [306]: s1 pd.DataFrame({a: [B, a, C], b: [1, 2, 3], c: [2, 3, 4]}).set\_index( .....: list(ab) .....: ) .....: In [307]: s1 Out[307]: c a b B 1 2 a 2 3 C 3 4 In [308]: s1.sort\_index(levela) Out[308]: c a b B 1 2 C 3 4 a 2 3 In [309]: s1.sort\_index(levela, keylambda idx: idx.str.lower()) Out[309]: c a b a 2 3 B 1 2 C 3 4 For information on key sorting by value, see value sorting. By values The Series.sort\_values() method is used to sort a Series by its values. The DataFrame.sort\_values() method is used to sort a DataFrame by its column or row values. The optional by parameter to DataFrame.sort\_values() may used to specify one or more columns to use to determine the sorted order. In [310]: df1 pd.DataFrame( .....: {one: [2, 1, 1, 1], two: [1, 3, 2, 4], three: [5, 4, 3, 2]} .....: ) .....: In [311]: df1.sort\_values(bytwo) Out[311]: one two three 0 2 1 5 2 1 2 3 1 1 3 4 3 1 4 2 The by parameter can take a list of column names, e.g.: In [312]: df1[[one, two, three]].sort\_values(by[one, two]) Out[312]: one two three 2 1 2 3 1 1 3 4 3 1 4 2 0 2 1 5 These methods have special treatment of NA values via the na\_position argument: In [313]: s[2] np.nan In [314]: s.sort\_values() Out[314]: 0 A 3 Aaba 1 B 4 Baca 6 CABA 8 cat 7 dog 2 NA 5 NA dtype: string In [315]: s.sort\_values(na\_positionfirst) Out[315]: 2 NA 5 NA 0 A 3 Aaba 1 B 4 Baca 6 CABA 8 cat 7 dog dtype: string Sorting also supports a key parameter that takes a callable function to apply to the values being sorted. In [316]: s1 pd.Series([B, a, C]) In [317]: s1.sort\_values() Out[317]: 0 B 2 C 1 a dtype: object In [318]: s1.sort\_values(keylambda x: x.str.lower()) Out[318]: 1 a 0 B 2 C dtype: object key will be given the Series of values and should return a Series or array of the same shape with the transformed values. For DataFrame objects, the key is applied per column, so the key should still expect a Series and return a Series, e.g. In [319]: df pd.DataFrame({a: [B, a, C], b: [1, 2, 3]}) In [320]: df.sort\_values(bya) Out[320]: a b 0 B 1 2 C 3 1 a 2 In [321]: df.sort\_values(bya, keylambda col: col.str.lower()) Out[321]: a b 1 a 2 0 B 1 2 C 3 The name or type of each column can be used to apply different functions to different columns. By indexes and values Strings passed as the by parameter to DataFrame.sort\_values() may refer to either columns or index level names. Build MultiIndex In [322]: idx pd.MultiIndex.from\_tuples( .....: [(a, 1), (a, 2), (a, 2), (b, 2), (b, 1), (b, 1)] .....: ) .....: In [323]: idx.names [first, second] Build DataFrame In [324]: df\_multi pd.DataFrame({A: np.arange(6, 0, -1)}, indexidx) In [325]: df\_multi Out[325]: A first second a 1 6 2 5 2 4 b 2 3 1 2 1 1 Sort by second (index) and A (column) In [326]: df\_multi.sort\_values(by[second, A]) Out[326]: A first second b 1 1 1 2 a 1 6 b 2 3 a 2 4 2 5 Note If a string matches both a column name and an index level name then a warning is issued and the column takes precedence. This will result in an ambiguity error in a future version. searchsorted Series has the searchsorted() method, which works similarly to numpy.ndarray.searchsorted(). In [327]: ser pd.Series([1, 2, 3]) In [328]: ser.searchsorted([0, 3]) Out[328]: array([0, 2]) In [329]: ser.searchsorted([0, 4]) Out[329]: array([0, 3]) In [330]: ser.searchsorted([1, 3], sideright) Out[330]: array([1, 3]) In [331]: ser.searchsorted([1, 3], sideleft) Out[331]: array([0, 2]) In [332]: ser pd.Series([3, 1, 2]) In [333]: ser.searchsorted([0, 3], sorternp.argsort(ser)) Out[333]: array([0, 2]) 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 [334]: s pd.Series(np.random.permutation(10)) In [335]: s Out[335]: 0 2 1 0 2 3 3 7 4 1 5 5 6 9 7 6 8 8 9 4 dtype: int64 In [336]: s.sort\_values() Out[336]: 1 0 4 1 0 2 2 3 9 4 5 5 7 6 3 7 8 8 6 9 dtype: int64 In [337]: s.nsmallest(3) Out[337]: 1 0 4 1 0 2 dtype: int64 In [338]: s.nlargest(3) Out[338]: 6 9 8 8 3 7 dtype: int64 DataFrame also has the nlargest and nsmallest methods. In [339]: df pd.DataFrame( .....: { .....: a: [-2, -1, 1, 10, 8, 11, -1], .....: b: list(abdceff), .....: c: [1.0, 2.0, 4.0, 3.2, np.nan, 3.0, 4.0], .....: } .....: ) .....: In [340]: df.nlargest(3, a) Out[340]: a b c 5 11 f 3.0 3 10 c 3.2 4 8 e NaN In [341]: df.nlargest(5, [a, c]) Out[341]: a b c 5 11 f 3.0 3 10 c 3.2 4 8 e NaN 2 1 d 4.0 6 -1 f 4.0 In [342]: df.nsmallest(3, a) Out[342]: a b c 0 -2 a 1.0 1 -1 b 2.0 6 -1 f 4.0 In [343]: df.nsmallest(5, [a, c]) Out[343]: a b c 0 -2 a 1.0 1 -1 b 2.0 6 -1 f 4.0 2 1 d 4.0 4 8 e NaN Sorting by a MultiIndex column You must be explicit about sorting when the column is a MultiIndex, and fully specify all levels to by. In [344]: df1.columns pd.MultiIndex.from\_tuples( .....: [(a, one), (a, two), (b, three)] .....: ) .....: In [345]: df1.sort\_values(by(a, two)) Out[345]: a b one two three 0 2 1 5 2 1 2 3 1 1 3 4 3 1 4 2 Copying The copy() method on pandas objects copies the underlying data (though not the axis indexes, since they are immutable) and returns a new object. Note that it is seldom necessary to copy objects. For example, there are only a handful of ways to alter a DataFrame in-place: Inserting, deleting, or modifying a column. Assigning to the index or columns attributes. For homogeneous data, directly modifying the values via the values attribute or advanced indexing. To be clear, no pandas method has the side effect of modifying your data; almost every method returns a new object, leaving the original object untouched. If the data is modified, it is because you did so explicitly. dtypes For the most part, pandas uses NumPy arrays and dtypes for Series or individual columns of a DataFrame. NumPy provides support for float, int, bool, timedelta64[ns] and datetime64[ns] (note that NumPy does not support timezone-aware datetimes). pandas and third-party libraries extend NumPys type system in a few places. This section describes the extensions pandas has made internally. See Extension types for how to write your own extension that works with pandas. See the ecosystem page for a list of third-party libraries that have implemented an extension. The following table lists all of pandas extension types. For methods requiring dtype arguments, strings can be specified as indicated. See the respective documentation sections for more on each type. Kind of Data Data Type Scalar Array String Aliases tz-aware datetime DatetimeTZDtype Timestamp arrays.DatetimeArray datetime64[ns, tz] Categorical CategoricalDtype (none) Categorical category period (time spans) PeriodDtype Period arrays.PeriodArray Period[freq] period[freq], sparse SparseDtype (none) arrays.SparseArray Sparse, Sparse[int], Sparse[float] intervals IntervalDtype Interval arrays.IntervalArray interval, Interval, Interval[numpy\_dtype], Interval[datetime64[ns, tz]], Interval[timedelta64[freq]] nullable integer Int64Dtype, (none) arrays.IntegerArray Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32, UInt64 nullable float Float64Dtype, (none) arrays.FloatingArray Float32, Float64 Strings StringDtype str arrays.StringArray string Boolean (with NA) BooleanDtype bool arrays.BooleanArray boolean pandas has two ways to store strings. object dtype, which can hold any Python object, including strings. StringDtype, which is dedicated to strings. Generally, we recommend using StringDtype. See Text data types for more. Finally, arbitrary objects may be stored using the object dtype, but should be avoided to the extent possible (for performance and interoperability with other libraries and methods. See object conversion). A convenient dtypes attribute for DataFrame returns a Series with the data type of each column. In [346]: dft pd.DataFrame( .....: { .....: A: np.random.rand(3), .....: B: 1, .....: C: foo, .....: D: pd.Timestamp(20010102), .....: E: pd.Series([1.0] 3).astype(float32), .....: F: False, .....: G: pd.Series([1] 3, dtypeint8), .....: } .....: ) .....: In [347]: dft Out[347]: A B C D E F G 0 0.035962 1 foo 2001-01-02 1.0 False 1 1 0.701379 1 foo 2001-01-02 1.0 False 1 2 0.281885 1 foo 2001-01-02 1.0 False 1 In [348]: dft.dtypes Out[348]: A float64 B int64 C object D datetime64[s] E float32 F bool G int8 dtype: object On a Series object, use the dtype attribute. In [349]: dft[A].dtype Out[349]: dtype(float64) If a pandas object contains data with multiple dtypes in a single column, the dtype of the column will be chosen to accommodate all of the data types (object is the most general). these ints are coerced to floats In [350]: pd.Series([1, 2, 3, 4, 5, 6.0]) Out[350]: 0 1.0 1 2.0 2 3.0 3 4.0 4 5.0 5 6.0 dtype: float64 string data forces an object dtype In [351]: pd.Series([1, 2, 3, 6.0, foo]) Out[351]: 0 1 1 2 2 3 3 6.0 4 foo dtype: object The number of columns of each type in a DataFrame can be found by calling DataFrame.dtypes.value\_counts(). In [352]: dft.dtypes.value\_counts() Out[352]: float64 1 int64 1 object 1 datetime64[s] 1 float32 1 bool 1 int8 1 Name: count, dtype: int64 Numeric dtypes will propagate and can coexist in DataFrames. If a dtype is passed (either directly via the dtype keyword, a passed ndarray, or a passed Series), then it will be preserved in DataFrame operations. Furthermore, different numeric dtypes will NOT be combined. The following example will give you a taste. In [353]: df1 pd.DataFrame(np.random.randn(8, 1), columns[A], dtypefloat32) In [354]: df1 Out[354]: A 0 0.224364 1 1.890546 2 0.182879 3 0.787847 4 -0.188449 5 0.667715 6 -0.011736 7 -0.399073 In [355]: df1.dtypes Out[355]: A float32 dtype: object In [356]: df2 pd.DataFrame( .....: { .....: A: pd.Series(np.random.randn(8), dtypefloat16), .....: B: pd.Series(np.random.randn(8)), .....: C: pd.Series(np.random.randint(0, 255, size8), dtypeuint8), [0,255] (range of uint8) .....: } .....: ) .....: In [357]: df2 Out[357]: A B C 0 0.823242 0.256090 26 1 1.607422 1.426469 86 2 -0.333740 -0.416203 46 3 -0.063477 1.139976 212 4 -1.014648 -1.193477 26 5 0.678711 0.096706 7 6 -0.040863 -1.956850 184 7 -0.357422 -0.714337 206 In [358]: df2.dtypes Out[358]: A float16 B float64 C uint8 dtype: object defaults By default integer types are int64 and float types are float64, regardless of platform (32-bit or 64-bit). The following will all result in int64 dtypes. In [359]: pd.DataFrame([1, 2], columns[a]).dtypes Out[359]: a int64 dtype: object In [360]: pd.DataFrame({a: [1, 2]}).dtypes Out[360]: a int64 dtype: object In [361]: pd.DataFrame({a: 1}, indexlist(range(2))).dtypes Out[361]: a int64 dtype: object Note that Numpy will choose platform-dependent types when creating arrays. The following WILL result in int32 on 32-bit platform. In [362]: frame pd.DataFrame(np.array([1, 2])) upcasting Types can potentially be upcasted when combined with other types, meaning they are promoted from the current type (e.g. int to float). In [363]: df3 df1.reindex\_like(df2).fillna(value0.0) df2 In [364]: df3 Out[364]: A B C 0 1.047606 0.256090 26.0 1 3.497968 1.426469 86.0 2 -0.150862 -0.416203 46.0 3 0.724370 1.139976 212.0 4 -1.203098 -1.193477 26.0 5 1.346426 0.096706 7.0 6 -0.052599 -1.956850 184.0 7 -0.756495 -0.714337 206.0 In [365]: df3.dtypes Out[365]: A float32 B float64 C float64 dtype: object DataFrame.to\_numpy() will return the lower-common-denominator of the dtypes, meaning the dtype that can accommodate ALL of the types in the resulting homogeneous dtyped NumPy array. This can force some upcasting. In [366]: df3.to\_numpy().dtype Out[366]: dtype(float64) astype You can use the astype() method to explicitly convert dtypes from one to another. These will by default return a copy, even if the dtype was unchanged (pass copyFalse to change this behavior). In addition, they will raise an exception if the astype operation is invalid. Upcasting is always according to the NumPy rules. If two different dtypes are involved in an operation, then the more general one will be used as the result of the operation. In [367]: df3 Out[367]: A B C 0 1.047606 0.256090 26.0 1 3.497968 1.426469 86.0 2 -0.150862 -0.416203 46.0 3 0.724370 1.139976 212.0 4 -1.203098 -1.193477 26.0 5 1.346426 0.096706 7.0 6 -0.052599 -1.956850 184.0 7 -0.756495 -0.714337 206.0 In [368]: df3.dtypes Out[368]: A float32 B float64 C float64 dtype: object conversion of dtypes In [369]: df3.astype(float32).dtypes Out[369]: A float32 B float32 C float32 dtype: object Convert a subset of columns to a specified type using astype(). In [370]: dft pd.DataFrame({a: [1, 2, 3], b: [4, 5, 6], c: [7, 8, 9]}) In [371]: dft[[a, b]] dft[[a, b]].astype(np.uint8) In [372]: dft Out[372]: a b c 0 1 4 7 1 2 5 8 2 3 6 9 In [373]: dft.dtypes Out[373]: a uint8 b uint8 c int64 dtype: object Convert certain columns to a specific dtype by passing a dict to astype(). In [374]: dft1 pd.DataFrame({a: [1, 0, 1], b: [4, 5, 6], c: [7, 8, 9]}) In [375]: dft1 dft1.astype({a: np.bool\_, c: np.float64}) In [376]: dft1 Out[376]: a b c 0 True 4 7.0 1 False 5 8.0 2 True 6 9.0 In [377]: dft1.dtypes Out[377]: a bool b int64 c float64 dtype: object Note When trying to convert a subset of columns to a specified type using astype() and loc(), upcasting occurs. loc() tries to fit in what we are assigning to the current dtypes, while [] will overwrite them taking the dtype from the right hand side. Therefore the following piece of code produces the unintended result. In [378]: dft pd.DataFrame({a: [1, 2, 3], b: [4, 5, 6], c: [7, 8, 9]}) In [379]: dft.loc[:, [a, b]].astype(np.uint8).dtypes Out[379]: a uint8 b uint8 dtype: object In [380]: dft.loc[:, [a, b]] dft.loc[:, [a, b]].astype(np.uint8) In [381]: dft.dtypes Out[381]: a int64 b int64 c int64 dtype: object object conversion pandas offers various functions to try to force conversion of types from the object dtype to other types. In cases where the data is already of the correct type, but stored in an object array, the DataFrame.infer\_objects() and Series.infer\_objects() methods can be used to soft convert to the correct type. In [382]: import datetime In [383]: df pd.DataFrame( .....: [ .....: [1, 2], .....: [a, b], .....: [datetime.datetime(2016, 3, 2), datetime.datetime(2016, 3, 2)], .....: ] .....: ) .....: In [384]: df df.T In [385]: df Out[385]: 0 1 2 0 1 a 2016-03-02 00:00:00 1 2 b 2016-03-02 00:00:00 In [386]: df.dtypes Out[386]: 0 object 1 object 2 object dtype: object Because the data was transposed the original inference stored all columns as object, which infer\_objects will correct. In [387]: df.infer\_objects().dtypes Out[387]: 0 int64 1 object 2 datetime64[ns] dtype: object The following functions are available for one dimensional object arrays or scalars to perform hard conversion of objects to a specified type: to\_numeric() (conversion to numeric dtypes) In [388]: m [1.1, 2, 3] In [389]: pd.to\_numeric(m) Out[389]: array([1.1, 2. , 3. ]) to\_datetime() (conversion to datetime objects) In [390]: import datetime In [391]: m [2016-07-09, datetime.datetime(2016, 3, 2)] In [392]: pd.to\_datetime(m) Out[392]: DatetimeIndex([2016-07-09, 2016-03-02], dtypedatetime64[ns], freqNone) to\_timedelta() (conversion to timedelta objects) In [393]: m [5us, pd.Timedelta(1day)] In [394]: pd.to\_timedelta(m) Out[394]: TimedeltaIndex([0 days 00:00:00.000005, 1 days 00:00:00], dtypetimedelta64[ns], freqNone) To force a conversion, we can pass in an errors argument, which specifies how pandas should deal with elements that cannot be converted to desired dtype or object. By default, errorsraise, meaning that any errors encountered will be raised during the conversion process. However, if errorscoerce, these errors will be ignored and pandas will convert problematic elements to pd.NaT (for datetime and timedelta) or np.nan (for numeric). This might be useful if you are reading in data which is mostly of the desired dtype (e.g. numeric, datetime), but occasionally has non-conforming elements intermixed that you want to represent as missing: In [395]: import datetime In [396]: m [apple, datetime.datetime(2016, 3, 2)] In [397]: pd.to\_datetime(m, errorscoerce) Out[397]: DatetimeIndex([NaT, 2016-03-02], dtypedatetime64[ns], freqNone) In [398]: m [apple, 2, 3] In [399]: pd.to\_numeric(m, errorscoerce) Out[399]: array([nan, 2., 3.]) In [400]: m [apple, pd.Timedelta(1day)] In [401]: pd.to\_timedelta(m, errorscoerce) Out[401]: TimedeltaIndex([NaT, 1 days], dtypetimedelta64[ns], freqNone) In addition to object conversion, to\_numeric() provides another argument downcast, which gives the option of downcasting the newly (or already) numeric data to a smaller dtype, which can conserve memory: In [402]: m [1, 2, 3] In [403]: pd.to\_numeric(m, downcastinteger) smallest signed int dtype Out[403]: array([1, 2, 3], dtypeint8) In [404]: pd.to\_numeric(m, downcastsigned) same as integer Out[404]: array([1, 2, 3], dtypeint8) In [405]: pd.to\_numeric(m, downcastunsigned) smallest unsigned int dtype Out[405]: array([1, 2, 3], dtypeuint8) In [406]: pd.to\_numeric(m, downcastfloat) smallest float dtype Out[406]: array([1., 2., 3.], dtypefloat32) As these methods apply only to one-dimensional arrays, lists or scalars; they cannot be used directly on multi-dimensional objects such as DataFrames. However, with apply(), we can apply the function over each column efficiently: In [407]: import datetime In [408]: df pd.DataFrame([[2016-07-09, datetime.datetime(2016, 3, 2)]] 2, dtypeO) In [409]: df Out[409]: 0 1 0 2016-07-09 2016-03-02 00:00:00 1 2016-07-09 2016-03-02 00:00:00 In [410]: df.apply(pd.to\_datetime) Out[410]: 0 1 0 2016-07-09 2016-03-02 1 2016-07-09 2016-03-02 In [411]: df pd.DataFrame([[1.1, 2, 3]] 2, dtypeO) In [412]: df Out[412]: 0 1 2 0 1.1 2 3 1 1.1 2 3 In [413]: df.apply(pd.to\_numeric) Out[413]: 0 1 2 0 1.1 2 3 1 1.1 2 3 In [414]: df pd.DataFrame([[5us, pd.Timedelta(1day)]] 2, dtypeO) In [415]: df Out[415]: 0 1 0 5us 1 days 00:00:00 1 5us 1 days 00:00:00 In [416]: df.apply(pd.to\_timedelta) Out[416]: 0 1 0 0 days 00:00:00.000005 1 days 1 0 days 00:00:00.000005 1 days gotchas Performing selection operations on integer type data can easily upcast the data to floating. The dtype of the input data will be preserved in cases where nans are not introduced. See also Support for integer NA. In [417]: dfi df3.astype(int32) In [418]: dfi[E] 1 In [419]: dfi Out[419]: A B C E 0 1 0 26 1 1 3 1 86 1 2 0 0 46 1 3 0 1 212 1 4 -1 -1 26 1 5 1 0 7 1 6 0 -1 184 1 7 0 0 206 1 In [420]: dfi.dtypes Out[420]: A int32 B int32 C int32 E int64 dtype: object In [421]: casted dfi[dfi 0] In [422]: casted Out[422]: A B C E 0 1.0 NaN 26 1 1 3.0 1.0 86 1 2 NaN NaN 46 1 3 NaN 1.0 212 1 4 NaN NaN 26 1 5 1.0 NaN 7 1 6 NaN NaN 184 1 7 NaN NaN 206 1 In [423]: casted.dtypes Out[423]: A float64 B float64 C int32 E int64 dtype: object While float dtypes are unchanged. In [424]: dfa df3.copy() In [425]: dfa[A] dfa[A].astype(float32) In [426]: dfa.dtypes Out[426]: A float32 B float64 C float64 dtype: object In [427]: casted dfa[df2 0] In [428]: casted Out[428]: A B C 0 1.047606 0.256090 26.0 1 3.497968 1.426469 86.0 2 NaN NaN 46.0 3 NaN 1.139976 212.0 4 NaN NaN 26.0 5 1.346426 0.096706 7.0 6 NaN NaN 184.0 7 NaN NaN 206.0 In [429]: casted.dtypes Out[429]: A float32 B float64 C float64 dtype: object Selecting columns based on dtype The select\_dtypes() method implements subsetting of columns based on their dtype. First, lets create a DataFrame with a slew of different dtypes: In [430]: df pd.DataFrame( .....: { .....: string: list(abc), .....: int64: list(range(1, 4)), .....: uint8: np.arange(3, 6).astype(u1), .....: float64: np.arange(4.0, 7.0), .....: bool1: [True, False, True], .....: bool2: [False, True, False], .....: dates: pd.date\_range(now, periods3), .....: category: pd.Series(list(ABC)).astype(category), .....: } .....: ) .....: In [431]: df[tdeltas] df.dates.diff() In [432]: df[uint64] np.arange(3, 6).astype(u8) In [433]: df[other\_dates] pd.date\_range(20130101, periods3) In [434]: df[tz\_aware\_dates] pd.date\_range(20130101, periods3, tzUSEastern) In [435]: df Out[435]: string int64 uint8 ... uint64 other\_dates tz\_aware\_dates 0 a 1 3 ... 3 2013-01-01 2013-01-01 00:00:00-05:00 1 b 2 4 ... 4 2013-01-02 2013-01-02 00:00:00-05:00 2 c 3 5 ... 5 2013-01-03 2013-01-03 00:00:00-05:00 [3 rows x 12 columns] And the dtypes: In [436]: df.dtypes Out[436]: string object int64 int64 uint8 uint8 float64 float64 bool1 bool bool2 bool dates datetime64[ns] category category tdeltas timedelta64[ns] uint64 uint64 other\_dates datetime64[ns] tz\_aware\_dates datetime64[ns, USEastern] dtype: object select\_dtypes() has two parameters include and exclude that allow you to say give me the columns with these dtypes (include) andor give the columns without these dtypes (exclude). For example, to select bool columns: In [437]: df.select\_dtypes(include[bool]) Out[437]: bool1 bool2 0 True False 1 False True 2 True False You can also pass the name of a dtype in the NumPy dtype hierarchy: In [438]: df.select\_dtypes(include[bool]) Out[438]: bool1 bool2 0 True False 1 False True 2 True False select\_dtypes() also works with generic dtypes as well. For example, to select all numeric and boolean columns while excluding unsigned integers: In [439]: df.select\_dtypes(include[number, bool], exclude[unsignedinteger]) Out[439]: int64 float64 bool1 bool2 tdeltas 0 1 4.0 True False NaT 1 2 5.0 False True 1 days 2 3 6.0 True False 1 days To select string columns you must use the object dtype: In [440]: df.select\_dtypes(include[object]) Out[440]: string 0 a 1 b 2 c To see all the child dtypes of a generic dtype like numpy.number you can define a function that returns a tree of child dtypes: In [441]: def subdtypes(dtype): .....: subs dtype.\_\_subclasses\_\_() .....: if not subs: .....: return dtype .....: return [dtype, [subdtypes(dt) for dt in subs]] .....: All NumPy dtypes are subclasses of numpy.generic: In [442]: subdtypes(np.generic) Out[442]: [numpy.generic, [[numpy.number, [[numpy.integer, [[numpy.signedinteger, [numpy.int8, numpy.int16, numpy.int32, numpy.int64, numpy.longlong, numpy.timedelta64]], [numpy.unsignedinteger, [numpy.uint8, numpy.uint16, numpy.uint32, numpy.uint64, numpy.ulonglong]]]], [numpy.inexact, [[numpy.floating, [numpy.float16, numpy.float32, numpy.float64, numpy.longdouble]], [numpy.complexfloating, [numpy.complex64, numpy.complex128, numpy.clongdouble]]]]]], [numpy.flexible, [[numpy.character, [numpy.bytes\_, numpy.str\_]], [numpy.void, [numpy.record]]]], numpy.bool\_, numpy.datetime64, numpy.object\_]] Note pandas also defines the types category, and datetime64[ns, tz], which are not integrated into the normal NumPy hierarchy and wont show up with the above function. previous Intro to data structures next IO tools (text, CSV, HDF5, ) On this page Head and tail Attributes and underlying data Accelerated operations Flexible binary operations Matching broadcasting behavior Missing data operations with fill values Flexible comparisons Boolean reductions Comparing if objects are equivalent Comparing array-like objects Combining overlapping data sets General DataFrame combine Descriptive statistics Summarizing data: describe Index of minmax values Value counts (histogramming) mode Discretization and quantiling Function application Tablewise function application Row or column-wise function application Aggregation API Aggregating with multiple functions Aggregating with a dict Custom describe Transform API Transform with multiple functions Transforming with a dict Applying elementwise functions Reindexing and altering labels Reindexing to align with another object Aligning objects with each other with align Filling while reindexing Limits on filling while reindexing Dropping labels from an axis Renaming mapping labels Iteration items iterrows itertuples .dt accessor Vectorized string methods Sorting By index By values By indexes and values searchsorted smallest largest values Sorting by a MultiIndex column Copying dtypes defaults upcasting astype object conversion gotchas Selecting columns based on dtype Show Source 2025, pandas via NumFOCUS, Inc. Hosted by OVHcloud. Created using Sphinx 8.1.3. Built with the PyData Sphinx Theme 0.14.4.