**Preface**

**What Is Data Science?**

Data science comprises three distinct and overlapping areas: the skills of a statistician who knows how to model and summarize datasets (which are growing ever larger); the skills of a computer scientist who can design and use algorithms to efficiently store, process, and visualize this data; and the domain expertise—what we might think of as “classical” training in a subject—necessary both to formulate the right questions and to put their answers in context.

A diagram of data science

Description automatically generated

**Why Python**

* NumPy for manipulation of homogeneous array-based data,
* Pandas for manipulation of heterogeneous and labeled data,
* SciPy for common scientific computing tasks,
* Matplotlib for publication-quality visualizations,
* IPython for interactive execution and sharing of code, Scikit-Learn for machine learning

**Jupyter: Beyond Normal Python**

**Getting Started in IPython and Jupyter**

* **IPython shell** for trying out short sequences of commands
* **Jupyter Notebook** for longer interactive analysis and for sharing content with others
* **Interactive development environments (IDEs)** like **Emacs** or **VSCode** for creating reusable Python packages.

**The IPython Shell**

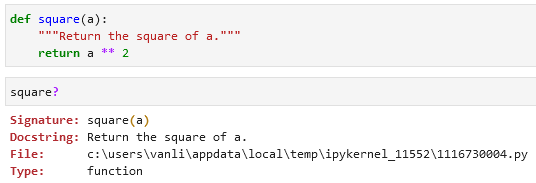
* Start the *IPython Shell* by typing ipython in the *Anaconda Prompt*.
* Launch Jupyter lab (?) $ jupyter lab

**Access documentation** with help() like help(len). The alternative is ? like len?. Get information on objects using ? like in the example below:

A screenshot of a computer

Description automatically generated

You can also get information on functions or other objects you create like in the example below which has a docstring (a description of the function):



?? provides the source code of the object you are curious about:

A screenshot of a computer

Description automatically generated

Use wildcard matching (character \*) to list every object in the namespace whose name ends with *Warning*:

A screenshot of a computer screen

Description automatically generated

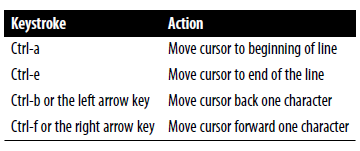
In the example below we area looking for a string method that contains the word *find* somewhere in its name:

A screenshot of a computer

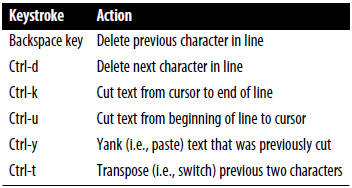
Description automatically generated

**Keyboard Shortcuts in the IPython Shell**

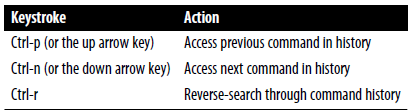
**Navigation Shortcuts**



**Text Entry Shortcuts**



**Command History Shortcuts**

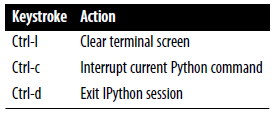


Use *Ctrl-r* to browse the search results. When you’re done press *Enter*.

A close up of text

Description automatically generated

**Miscellaneous Shortcuts**



**Enhanced Interactive Features**

**IPython Magic Commands**

Magic commands are prefixed by the % character.

* **Line magics** are denoted by a single % prefix. They operate on a **single line of input**.
* **Cell magics** are denoted by a double %% prefix and operate on **multiple lines of input**.
* Magic commands documentation %magic.
* Quick and simple list of all available magic functions %lsmagic.

If you have a *.py* file you can execute it / run its script in IPython using %run filename.py:

A screenshot of a computer screen

Description automatically generated

To time code execution use %timeit or %%timeit for multiple lines of code:

A screenshot of a computer

Description automatically generated

You can check you input and output history using In and Out codes. You can also select specific steps like print(In[1]).

A number and numbers on a white background

Description automatically generated

A white background with red text

Description automatically generated

A black text on a white background

Description automatically generated

A screenshot of a computer program

Description automatically generated

An alternative to Out[20] is \_20:



Use print(\_) to get access to the previous output:

A screenshot of a phone

Description automatically generated

You can also use print(\_\_) and print(\_\_\_) to get access to the second/third-to-last outputs.

If you want to suppress you output you can do it using ; at the end of the line:

A white background with black text

Description automatically generated

Use %history to get an overview of your commands, use %history -n to get an numbered overview of your commands. To select a command/commands add a number like 3-5:

A black screen with white text

Description automatically generatedA screenshot of a computer

Description automatically generated

Other useful commands are %rerun (re-execute some portion of the command history) and %save (saves some set of the command history to a file).

**Quick Introduction to the Shell**

Some shell commands:



A screenshot of a computer program

Description automatically generated

echo, pwd, ls, cd, mkdir, mv

A black text on a white background

Description automatically generated

A screenshot of a computer program

Description automatically generated

!echo

A screenshot of a computer

Description automatically generated!pwd !cd

A screenshot of a computer program

Description automatically generated%cd mkdir ls cp rm -r

**Debugging and Profiling**

**Controlling Exceptions using %xmode**

There are 3 formats for errors: Plain, Context, and Verbose.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

The last provides the most information but can be bulky.

**To launch a debugger** write %debug.

**Launch the Sdebugger automatically** whenever an exception is raised using

%xmode Plain

%pdb on

A computer code with text

Description automatically generated

**Partial list of debugging commands:**

A screenshot of a computer program

Description automatically generatedl(ist) h(elp) q(uit) c(ontinue) n(ext) <enter> p(rint) s(tep) r(eturn)

**Profiling and Timing Code**

%time Time the execution of a single statement

%timeit Time repeated execution of a single statement for more accuracy

%prun Run code with the profiler

%lprun Run code with the line-by-line profiler

%memit Measure the memory use of a single statement

%mprun Run code with the line-by-line memory profiler

%timeit

A screenshot of a computer code

Description automatically generated

%time

A screenshot of a computer program

Description automatically generated

%prun

A screenshot of a computer

Description automatically generated

A close up of text

Description automatically generated

%lprun

A screenshot of a computer

Description automatically generated%load\_ext line\_profiler %lprun -f

A close up of black text

Description automatically generated

%memit (is a memory-measuring equivalent of %timeit) and %mprun (memory-measuring equivalent of %lprun)

A white rectangular sign with black text

Description automatically generated pip install memory\_profiler

A white rectangular object with black text

Description automatically generated%load\_ext memory\_profiler

A white rectangular object with green text

Description automatically generated with medium confidence %memit

A close-up of a text

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated%%file %mprun -f

**NumPy**

In some ways, NumPy arrays are like Python’s built-in *list* type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size.

Check NumPy version: numpy.\_\_version\_\_

**Fixed-Type Arrays in Python**

Example of an array in Python:

A screenshot of a computer code

Description automatically generatedimport array, array.array



Python arrays objects provide efficient storage. NumPy adds to this efficient operations on that data.

**Creating Arrays from Python Lists**

NumPy arrays can only contain data of the same type. This is unlike Python lists.

NumPy arrays can be multidimensional. Python lists are always one-dimensional sequences.

A screenshot of a computer program

Description automatically generated

**Creating Arrays from Scratch**

A close up of words

Description automatically generatedA screenshot of a computer program

Description automatically generated

A screenshot of a computer code

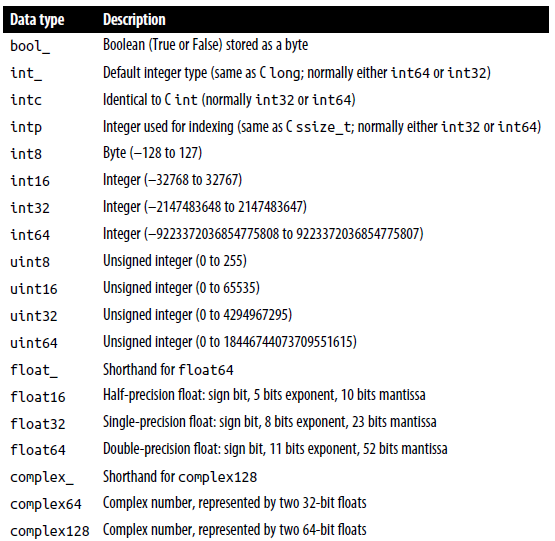
Description automatically generated

**Specify data type when creating an array using dtype=’int16’ or dtype=np.int16:**

A screenshot of a computer

Description automatically generated

**Standard NumPy data types:**



**NumPy Array Attributes**

A screenshot of a computer code

Description automatically generated

ndim, shape, size, dtype, default\_rng(seed= integers

**Array Slicing: Accessing Subarrays**

Follow this logic to access a slice of an array x:

x[start:stop:step]

If any of these are unspecified, they default to the values start=0, stop=<size of dimension>, step=1.



A screenshot of a computer code

Description automatically generated

A white background with black text

Description automatically generatedreverse an array [::-1]

A white background with numbers and symbols

Description automatically generatedreverse a multiarray [::-1, ::-1]

 first column of a multiarray [0, :]

 first row of a multiarray [0] or [0, :]



**Subarrays as no-copy views**

Unlike Python list slices, NumPy array slices are returned as views rather than copies of the array data.

In the example below if we modify the subarray *x2\_sub* the original array *x2* will also change.

A screenshot of a phone

Description automatically generated

It can be advantageous when working with large datasets. We can access and process pieces of these datasets without the need to copy the underlying data buffer.

If you want to create a copy that can be changed without affecting the original data use the .copy() code:

A screenshot of a computer

Description automatically generated

**Reshaping of arrays**

Use .reshape() to put if you want to put e.g. numbers from 1 to 9 in a 3 × 3 grid:

A screenshot of a computer code

Description automatically generated

Note that the size of the array should match the size of the reshaped array.

You can also use .reshape() to convert a one-dimensional array into…

…a two-dimensional row matrix: …or a column matrix:

A screenshot of a computer code

Description automatically generated A screenshot of a computer code

Description automatically generated

You can achieve the same using np.newaxis in the slicing index:

A close-up of a web page

Description automatically generated A screenshot of a computer code

Description automatically generated

**Array concatenation (combine multiple arrays into one)**

Use np.concatenate, np.vstack, and np.hstack to join two arrays.

A screenshot of a computer code

Description automatically generated A screenshot of a computer code

Description automatically generated

Concatenate a two-dimensional array along the first axis: …and among the second axis:

A screenshot of a computer code

Description automatically generated A screenshot of a computer

Description automatically generated

When you have arrays of mixed dimensions you can use np.vstack and np.hstack:

A screenshot of a phone

Description automatically generated

For higher-dimensional arrays np.dstack will stack arrays along the third axis.

**Array splitting (split a single array into multiple arrays)**

To split arrays use functions np.split, np.hsplit, np.vsplit.

Using np.split(array\_name, [splitting\_location(s)]):

A screenshot of a math problem

Description automatically generated

Splitting using np.hsplit and np.vsplit:

A screenshot of a computer program

Description automatically generated

For higher-dimensional arrays np.dsplit will split arrays along the third axis.

**NumPy universal functions (ufuncs)**

To make computation on NumPy arrays fast use vectorized operations that are generally implemented through *universal functions (ufuncs)*.

There are *two types of ufuncs:*

* *Unary ufuncs* operate on a single input
* *Binary ufuncs* operate on two inputs.

**Arithmetic operators examples in NumPy:**

A screenshot of a computer

Description automatically generated A math equations on a white background

Description automatically generated

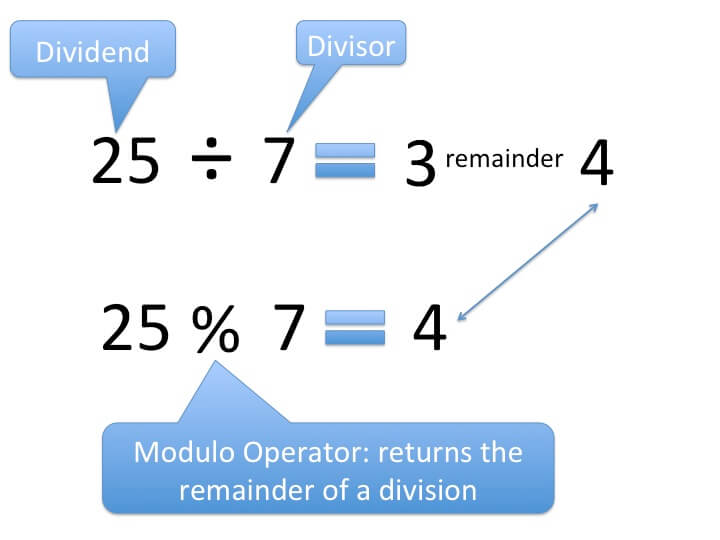
- is negation (turn *1* into *-1*), % is an operator for *modulus \**.

These arithmetic operations are convenient wrappers around specific ufuncs built into NumPy. For example, the + operator is a wrapper for the np.add() ufunc:

A screenshot of a computer code

Description automatically generated

*\* Modulus:*



**Arithmetic operators implemented in NumPy:**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Equivalent ufunc** | **Description** |
| + | np.add | Addition (e.g., 1 + 1 = 2) |
| - | np.subtract | Subtraction (e.g., 3 - 2 = 1) |
| - | np.negative | Unary negation (e.g., -2) |
| \* | np.multiply | Multiplication (e.g., 2 \* 3 = 6) |
| / | np.divide | Division (e.g., 3 / 2 = 1.5) |
| // | np.floor\_divide | Floor division (e.g., 3 // 2 = 1) |
| \*\* | np.power | Exponentiation (e.g., 2 \*\* 3 = 8) |
| % | np.mod | Modulus/remainder (e.g., 9 % 4 = 1) |

To make values absolute (e.g. from -2 to 2) use abs(name), np.absolute(name) or np.abs(name):

A screenshot of a computer

Description automatically generated

**Trigonometric functions**

Functions like np.sin, np.cos, np.tan:

A screenshot of a computer

Description automatically generated

Inverse trigonometric functions like np.arcsin, np.arccos, np.arctan:

A screenshot of a math program

Description automatically generated

**Exponents and logarithms**

Exponentials in NumPy like np.exp, np.exp2, np.power:

A screenshot of a math program

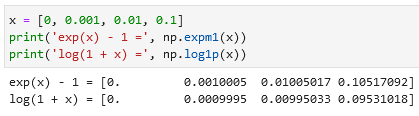
Description automatically generated

Inverse of the exponentials or the logarithms are computed using np.log (natural logarithm), np.log2 (base-2 logarithm), and np.log10 (base-10 logarithm):

A screenshot of a computer program

Description automatically generated

Specialized versions useful for maintaining precision with very small input (np.expm1 and np.log1p). When x is very small, these functions give more precise values than if the raw np.log or np.exp were to be used.



**Specialized ufuncs**

You can use scipy special for specialized functions. For example gamma functions (generalized factorials) and related functions (special.gamma, special.gammaln, special.beta):

A screenshot of a computer

Description automatically generated

Error functions (integral of Gaussian), its complement, and its inverse (special.erf, special.erfc, special.erfinv):

A screenshot of a computer code

Description automatically generated

**Specifying output of an array**

For large calculations, it is sometimes useful to be able to specify the array where the result of the calculation will be stored. For all ufuncs, this can be done using the out argument of the function:

A screenshot of a computer

Description automatically generated

This can even be used with *array views*. For example, we can write the results of a computation to every other element of a specified array (in the example below every second):

A screenshot of a math equation

Description automatically generated

If we had instead written y[::2] = 2 \*\* x, this would have resulted in the creation of a temporary array to hold the results of 2 \*\* x, followed by a second operation copying those values into the y array. This doesn’t make much of a difference for such a small computation, but **for very large arrays the** **memory savings from careful use of the out argument can be significant**.

**Aggregations**

Use the reduce function to reduce an array with a particular operation.

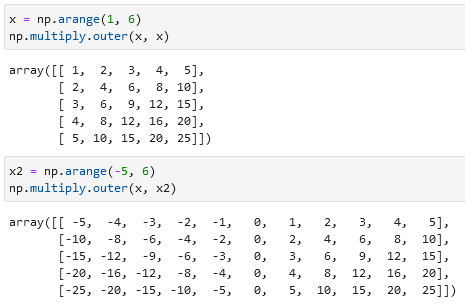
You can use np.add.reduce, np.multiply.reduce to sum or multiply an array. If you want to store intermediate results use np.add.accumulate or np.multiply.accumulate.

A screenshot of a computer

Description automatically generated

**Outer products**

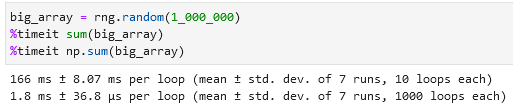
Compute the output of all pairs of two inputs using np.multiply.output(input\_1, input\_2):



**Aggregations in NumPy: min, max, and everything in between**

**Summing the Values in an Array**

You can sum using Python like sum(x) or you can use NumPy’s np.sum(x). Because NumPy executes the operation in a compiled code, **NumPy’s version of the operation is computed much more quickly**:



However, sum(x) and np.sum(x) are not identical. Their optional arguments have different meanings. Python’s *sum(x, 1)* initializes the sum at *1* (meaning that if your sum(x) is equal to *5*, your sum(x, 1 or 10) will be equal to *5 + 1 or 10 = 6 or 15*), while np.sum(x, 1) sums along *axis 1*.Also, np.sum is aware of multiple array dimensions.

**Minimum and maximum (min, max)**

Same as with summing you can use Python’s min(x) or NumPy’s np.min(x) or max(x) / np.max(x). An alternative is x.min() / x.max() / x.sum(). Same as with summing, **NumPy’s version is faster**.

**Multidimensional Aggregates**

Finding the minimum value within each column/row by specifying axis=0 / axis=1:

A screenshot of a computer

Description automatically generated

The axis keyword specifies the dimension of the array that will be *collapsed*, rather than the dimension that will be returned. So, specifying axis=0 means that axis 0 will be collapsed: for two-dimensional arrays, values within each column will be aggregated.

**Other aggregation functions**

Most additional aggregation functions have a NaN-safe counterpart that computes the result while ignoring missing values.

A table of words

Description automatically generated

A white background with black text

Description automatically generated

np.sum, np.prod, np.mean, np.std, np.var, np.min, np.max, np.argmin, np.argmax, np.median, np.percentile, np.any, np.all



A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

**Broadcasting**

Broadcasting allows these types of binary operations to be performed on arrays of different sizes—for example, we can just as easily add a scalar (think of it as a zerodimensional array) to an array:

A screenshot of a computer

Description automatically generated

We can think of this as an operation that stretches or duplicates the value 5 into the array [5, 5, 5], and adds the results.

**Broadcasting is stretching an array** across a (in the example below) second dimension:



A screenshot of a phone

Description automatically generated

You can also stretch or broadcast both arrays to match a common shape:

A screenshot of a computer program

Description automatically generated

**Visualization of broadcasting:**

A group of cubes with numbers

Description automatically generated

**Rules of broadcasting**

1. If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
2. If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
3. If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

Rule 1 example: Rule 2 example:

A screenshot of a computer program

Description automatically generated A white paper with black text and white text

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generated

**Using broadcasting for centering an array**

You can subtract from an array the mean of that array.

|  |  |
| --- | --- |
| **Array:** | **Centered (difference to mean):** |
| 34 | -4 |
| 65 | 27 |
| 23 | -15 |
| 76 | 38 |
| 22 | -16 |
| 43 | 5 |
| 63 | 25 |
| 32 | -6 |
| 23 | -15 |
| 27 | -11 |
| 5 | -33 |
|  |  |
| **Mean:** |  |
| 38 |  |

A screenshot of a computer program

Description automatically generated

A white background with black text

Description automatically generated

**Comparisons, Masks, and Boolean Logic**

**Comparison Operators as Ufuncs**

As in the case of arithmetic operators, the comparison operators are implemented as ufuncs in NumPy; for example, when you write x < 3, internally NumPy uses np.less(x, 3). These operators will work on arrays of any size and shape.

A black and white background with black text

Description automatically generatednp.equal, np.less, np.greater, np.not\_equal, np.less\_equal, np.greater\_equal

Example:

A screenshot of a computer code

Description automatically generated

In each case, the result is a Boolean array, and NumPy provides a number of straight-forward patterns for working with these Boolean results.

**Working with Boolean Arrays**

**Counting Entries**

To count the number of True entries you can use np.count\_nonzero:

A screenshot of a computer

Description automatically generated

A white rectangular box with text

Description automatically generated with medium confidence

**Alternatively you can use np.sum (*False* is *0* and *True* is *1*).** With np.sum you can sum along rows or columns.

A white rectangular box with black text

Description automatically generated A white rectangular box with black text

Description automatically generated ← e.g. in row 1 there are 3 values below 6, in row 2 2 values, in row 3 3.

If you want to count the number of True entries that fulfill multiple conditions like *x > 10 AND x < 20* you can use symbols like &, |, ^, and ~:

A white rectangular object with text

Description automatically generated with medium confidence

A white rectangular object with text

Description automatically generated ← same result in a different manner

Note that parentheses are important. Without them this code with result in an error.

**If you want to check if an array has values that fulfill a specific condition you can use np.any or np.all.**

A screenshot of a cell phone

Description automatically generated

You can also use np.any and np.all along a particular axis:

A screenshot of a computer

Description automatically generated

**Boolean operators and their equivalent ufuncs**

|  |  |
| --- | --- |
| **Operator** | **Equivalent ufunc** |
| & | np.bitwise\_and |
| ^ | np.bitwise\_xor |
| | | np.bitwise\_or |
| ~ | np.bitwise\_not |

Examples:

A screenshot of a computer

Description automatically generated

**Boolean arrays as masks**

Selecting values from an array using Boolean operators is called a masking operation. Example:

A screenshot of a computer program

Description automatically generated

A one-dimensional array is returned with all the values that meet this condition.

**Examples of Boolean arrays as masks:**

A screenshot of a computer

Description automatically generated

**Comment on using and/or instead of &/|**

The difference is this: and and or operate on the object as a whole, while & and | operate on the elements within the object.

and and or perform a single Boolean evaluation on an entire object, while & and | perform multiple Boolean evaluations on the content (the individual bits or bytes) of an object. **For Boolean NumPy arrays, the latter (& and |) is nearly always the desired operation.**

Using and and or instead of &/| will simply result in an error:

A screenshot of a computer

Description automatically generated

**Fancy indexing / vectorized indexing**

To access values with specific indexes you can use this:

A screenshot of a computer

Description automatically generated

When using arrays of indices, the shape of the result reflects the shape of the index arrays rather than the shape of the array being indexed. In the example below x[ind] returns the same form as of *ind* but instead of index coordinates (*3, 7,* etc.) it returns values from *x*.

A screenshot of a computer code

Description automatically generated

A screenshot of a white text

Description automatically generated

**Combined indexing**

A screenshot of a computer

Description automatically generated

A screenshot of a computer code

Description automatically generated

**Modifying values with fancy indexing**

A screenshot of a computer

Description automatically generated

An alternative to np.add.at is reduceat.

**Example: Binning data**

You could use these ideas to efficiently do custom binned computations on data. For example, imagine we have 100 values and would like to quickly find where they fall within an array of bins. We could compute this using ufunc.at like this:

A screen shot of a graph

Description automatically generated

Alternatively using Matplotlib:

A graph on a grid

Description automatically generated

The difference between these two options is in the processing speed. Matplotlib is slower than the custom version for a small dataset and faster for a larger.

**Sorting arrays**

**In Python**

Python’s built-in functions to sort lists and other iterable objects: sorted(x) (returns a copy) and x.sort() (acts in-place). Two examples:

A screenshot of a computer

Description automatically generated



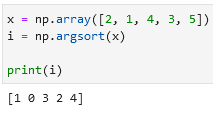
**In NumPy**

The np.sort function is analogous to sorted(). It will return a sorted copy of an array. Example:

A screenshot of a computer

Description automatically generated

np.argsort() returns the indices of the sorted elements:



You can then use these indices to construct the sorted array using fancy indexing:

A screenshot of a computer

Description automatically generated

Sorting along rows or columns is possible by adding axis=0 for sorting columns or axis=1 for sorting rows. Note that this operation will treat each row or column as an independent array and any relationship between the row or column values will be lost.

A screenshot of a computer

Description automatically generated

**You can also find e.g. 3 smallest values in an array using np.partition.** It will return these smallest values in arbitrary order followed by other values also in arbitrary order.

A screenshot of a computer

Description automatically generated

**Checking the diagonal of a matrix**

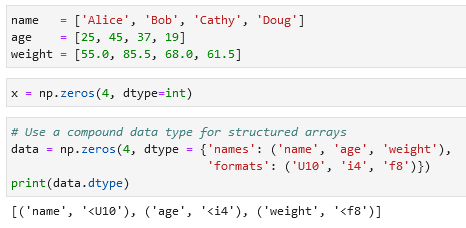
You can check the diagonal values of a matrix using x.diagonal():

A screenshot of a computer

Description automatically generated

**NumPy’s structured arrays and record arrays**

Creating an empty container array:



* U10 = Unicode string of maximum length 10
* i4 = 4-byte (i.e. 32-bit) integer
* f8 = 8-byte (i.e. 64-bit) float

Filling the empty array with data in a one structured array:

A screenshot of a computer

Description automatically generated

Filtering the data:

A screenshot of a computer

Description automatically generated

**Exploring structured array creation — specifying data type**

A screenshot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generated

**NumPy data types:**

A table with text and numbers

Description automatically generated

'b' Byte np.dtype('b') 'i' Signed integer np.dtype('i4') == np.int32 'u' Unsigned integer np.dtype('u1') == np.uint8 'f' Floating point np.dtype('f8') == np.int64 'c' Complex floating point np.dtype('c16') == np.complex128 'S', 'a' String np.dtype('S5') 'U' Unicode string np.dtype('U') == np.str\_ 'V' Raw data (void) np.dtype('V') == np.void

**Accessing data using record arrays**

You can use view(np.recarray) to access columns:

A screenshot of a computer program

Description automatically generated

Note that record arrays are slightly slower than a simple data['age'].

**Pandas**

**Pandas has 3 fundamental data structures: the Series, DataFrame, and Index.**

**A series is a one-dimensional array of indexed data. It can be created from a list or array. Example of a series:**

**A white background with black text

Description automatically generated**

A screenshot of a computer code

Description automatically generated

Use .values and .index to access values and indexes of a series.

You can assign indexes that you want (different from NumPy arrays):

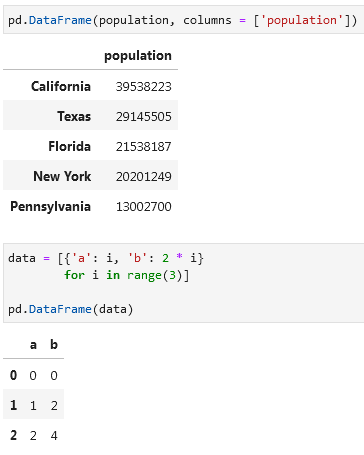
A screenshot of a computer

Description automatically generated

A screenshot of a computer code

Description automatically generated

**Ways to create a DataFrame**

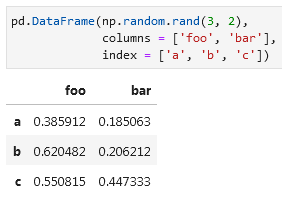


A screenshot of a computer

Description automatically generated

A screenshot of a computer

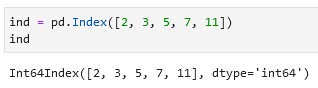
Description automatically generated



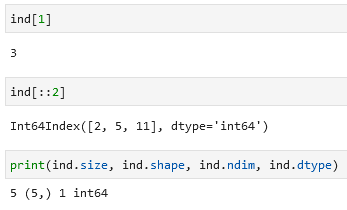
A screenshot of a computer

Description automatically generated

**The Pandas Index object**



Difference between Index objects and NumPy arrays is that the indices are immutable — they cannot be modified via the normal means like ind[1] = 0. This will cause an error.



Using Index you can get values that are present in both sets, make a union of all values in both sets, and show only unique values in both sets. Use .intersection(), .union(), and .symmetric\_difference() for these operations. Examples:

A screenshot of a computer

Description automatically generated

**Data Indexing and Selection**

**Accessing indices or values in a Pandas Series:**

A screenshot of a computer

Description automatically generated in, .keys(), .items()

**Add/change values to a series:**

A screenshot of a computer

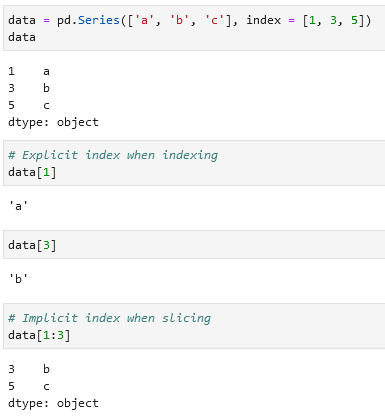
Description automatically generated

**Slicing, masking, and fancy indexing in a Series:**

A screenshot of a computer code

Description automatically generated

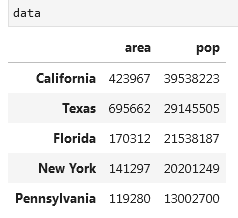
**.loc references the index provided (by e.g. you) while .iloc uses the standard index 0, 1, 2, 3, etc.:**



A screenshot of a computer code

Description automatically generated

**Data selection in DataFrames**



This way of selecting data is preferred:

A screenshot of a computer

Description automatically generated

Over this one:

A screenshot of a computer

Description automatically generated

Adding columns created from given columns:

A screenshot of a computer

Description automatically generated

Example with .iloc and .loc:

A screenshot of a computer

Description automatically generated

Getting all values from a DataFrame using .values and transforming the table using .T:

A screenshot of a computer

Description automatically generated

Other examples:

A screenshot of a computer

Description automatically generated

**Adding two Series/DataFrames**

If you add two series with different indices you will get NaN values:

A screenshot of a computer code

Description automatically generated

To avoid it use .add(fill\_value = 0). It will use that specified value in place of missing entries. Example:

A screenshot of a computer code

Description automatically generated

Same is applies for two DataFrames:

A screenshot of a phone

Description automatically generated A screenshot of a computer

Description automatically generated

A screenshot of a math test

Description automatically generated

In this example the fill\_value is the mean value:

A screenshot of a computer

Description automatically generated

**Mapping between Python operators and Pandas methods:**

|  |  |
| --- | --- |
| **Python operator** | **Pandas method(s)** |
| + | add |
| - | sub, subtract |
| \* | mul, multiply |
| / | truediv, div, divide |
| // | floordiv |
| % | mod |
| \*\* | pow |

**Subtract a row/column from a DataFrame**

A screenshot of a computer

Description automatically generated

**Handling Missing Data in pandas**

**None as blank value**

Example:

A screenshot of a computer code

Description automatically generated

Downside is that it is counted as a Python object data type. Therefore processing times will be slow. In addition arithmetic operations are not supported with None.

**nan: Missing Numerical Data**

Example (np.nan):

A screenshot of a computer code

Description automatically generated

It supports fast operations but any arithmetic operations with nan will lead to a nan:

A screenshot of a computer

Description automatically generated

You can, of course, use special functions (e.g. np.nansum(), np.nanmin(), npnanmax()) that ignore nan:

A close-up of a computer screen

Description automatically generated

The main downside of nan is that it is specifically a floating-point value; there is no equivalent nan value for integers, strings, or other types.

**nan and None in pandas**

Both are transformed into a NaN value and the array gets a float64 data type:

A screenshot of a computer

Description automatically generated

If a None value is added to a Series then the data type changes automatically to float64:

A screenshot of a computer program

Description automatically generated

**Pandas handling of NAs by type:**

A black and white text

Description automatically generatedfloat64, object, np.nan

**Pandas nullable dtypes**

Nullable dtypes are distinguished from regular dtypes by capitalization of their names, e.g. pd.Int32 vs np.int32.

Example that includes all three available markers of missing data (None, np.nan,and pd.NA):

A screen shot of a computer

Description automatically generated

**Operation on null values in pandas**

There are several methods for detecting, removing, and replacing null values in pandas:

* isnull
* notnull
* dropna
* fillna

Example with .isnull() to see what values are null:

A screenshot of a computer program

Description automatically generated

Example with .notnull() to filter values that are not null:

A screenshot of a computer

Description automatically generated

Example with .dropna() which removes null values:

A screen shot of a computer

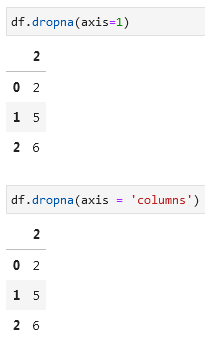
Description automatically generated

If you use .dropna() on a DataFrame you can drop entire rows or columns with null values. By default drops all rows where there is at least one null value.

A screenshot of a computer

Description automatically generated

To drop columns with null values specify axis=1 or axis = 'columns':



If you want to drop only rows or columns where *every* value is null instead of just *one*, you need to add how='all' (by default this field is how='any').

A screenshot of a computer

Description automatically generated

If you want to keep only rows or columns that have a minimum number of values that are not equal to null, then you can specify this using the thresh = \_ parameter. In the example below only rows that have 3 or more non-null values will be kept:

A screenshot of a math equation

Description automatically generated

**Filling null values / replacing null values**

A screenshot of a computer code

Description automatically generated

Depending on the situation, you may wish to keep the null values but replace them with something else. You can fill them with a number like in the example below. Function .fillna() is used for it.

A screenshot of a computer

Description automatically generated

Using .fillna(method = 'ffill') you can copy the previous value to the empty cell:

A screenshot of a computer program

Description automatically generated

Alternatively you can use the bfill method to copy the next value to the empty cell:

A screenshot of a computer program

Description automatically generated

If you use a ffill or bfill on a DataFrame you can specify an axis along which the fills should take place. Note that if a previous value is not available during a fill, the null value will remain.

A screenshot of a computer

Description automatically generated

**Hierarchical Indexing**

If you want to create a multiply indexed series (an index with multiple layers) you can use this approach, but it is not efficient:

A screenshot of a computer program

Description automatically generated

A much better one is to use pd.MultiIndex.from\_tuples() with .reindex():

A screenshot of a computer code

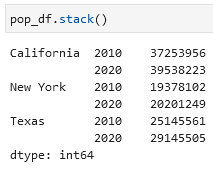
Description automatically generated

In case you want to unpivot a multiply indexed Series into a DataFrame you can use .unstack():

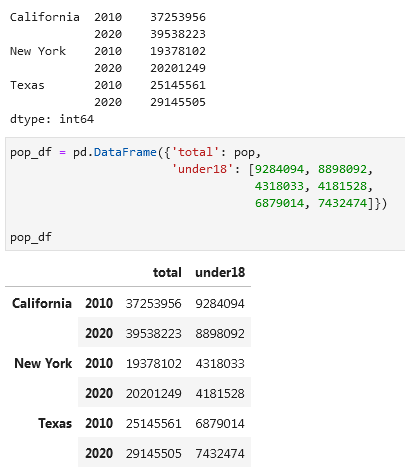
A screenshot of a computer

Description automatically generated

If you want to pivot a DataFrame you can use .stack():



If you want to add a column to your DataFrame:



Calculating arithmetically the percentage of underage population per state per year using .unstack():

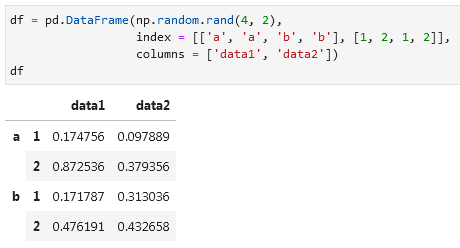
A screenshot of a computer

Description automatically generated

**Methods of MultiIndex creation**

**Automatic**

If you pass a list of two or more index arrays to the constructor you will get a multiply indexed Series or a DataFrame:



If you pass a dictionary with appropriate tuples as keys, pandas will automatically recognize this and use a MultiIndex by default:

A screenshot of a computer

Description automatically generated

**Explicit**

Use pd.MultiIndex.from\_arrays or pd.MultiIndex.from\_product or pd.MultiIndex to create an explicit multiple index:



**Giving names to multiple indexes**

You can pass names to the multiple indexes that you have using .index.names = []:

A screenshot of a computer

Description automatically generated

**MultiIndex for columns**

A screenshot of a computer

Description automatically generated

**Accessing values that have multiple indexes**

To access a value that has multiple indexes write these indexes one after another:

A screenshot of a computer

Description automatically generated

Accessing one index:

A screenshot of a computer

Description automatically generated

Selecting a specific part. Note that MultiIndex should be sorted to use partial slicing.

A screenshot of a computer

Description automatically generated

If you want to filter on the second/third/etc. index and not on first:

A screenshot of a computer code

Description automatically generated

Selection based on Boolean masks or simply filtering values that fit your desired result:

A screenshot of a computer

Description automatically generated

Selection based on fancy indexing or simply selecting specific rows:

A screenshot of a computer

Description automatically generated

**Examples with multiple indexes in rows and columns**

A screenshot of a computer

Description automatically generated

If you want to use index slicing use pd.IndexSlice:

A screenshot of a computer

Description automatically generated

**Sorting indexes**

Many of the MultiIndex slicing operations will fail if the index is not sorted. For sorting indexes you can use .sort\_index() or sortlevel:

A screenshot of a computer

Description automatically generated

**Unstacking multilevel indexes (unpivoting)**

Same as with a normal Series/DataFrame you use .stack()/.unstack() to unpivot a table. To specify an index add (level = 0) or e.g. (level = 1):

A screenshot of a computer

Description automatically generated

**Turning an index into a column**

You can turn an index into a column by using .reset\_index(name = \_) function. In the name field you specify the name of the column that will have data in it:

A screenshot of a computer

Description automatically generated

You can also turn columns into multiple indexes using .set\_index():

A screenshot of a computer

Description automatically generated

**Combining Datasets: concat and append**

Concatenation of Series and DataFrame is similar to concatenation of NumPy arrays, which can be done using the np.concatenate() function. If you have a multidimensional array you need to specify the axis along which you want to concatenate.

A screenshot of a computer

Description automatically generated

Combining datasets using pd.concat():

A screenshot of a computer program

Description automatically generated

When used on DataFrames pd.concat() will by default concatenate rows (will place values below the first DataFrame). You can specify axis = 'columns' (or axis = 1) to concatenate columns.

**Handing duplicate indexes**

One important difference between np.concatenate and pd.concat is that Pandas concatenation preserves indexes, even if the result will lead to duplicates.

A screenshot of a computer

Description automatically generated

To check if your datasets contain duplicate indexes you can add the verify\_integrity = True option. You will get an error if there are duplicates.

A white rectangular box with black text

Description automatically generated

If you want to ignore the indexes of your datasets you can add the ignore\_index = True option. You will get a standard index (0, 1, 2, 3, etc.) in your concatenated dataset.

A screenshot of a computer

Description automatically generated

Another option with duplicate indexes is to add an extra index that will specify the origin of a dataset. You use for that the keys option.

A screenshot of a computer

Description automatically generated

**Concatenation with joins**

If you want to concatenate datasets that have different columns along with some columns that both datasets have in common you will get this:

A screenshot of a computer

Description automatically generated

Columns that are unique to each dataset will be filled with null values where appropriate. This is because the default join takes everything from both datasets (option join='outer'). You can change it to inner to only concatenate columns that overlap in both datasets:

A screenshot of a computer

Description automatically generated

To get more control over which columns are dropped you can use the .reindex() function (не до конца понял, как оно работает):

A screenshot of a computer

Description automatically generated

Alternative to pd.concat([df1, df2]) is df1.append(df2). But this function will be removed from pandas in the future versions, so use pd.concat.