Python Data Science

NumPy & pandas

NumPy and pandas

https://github.com/squillero/python-lectures/

september 2023 **version 0.5**

Python Data Science

numpy

NumPy

- With **conda**, install **numpy** either from the default channel or **conda-forge**
- Otherwise, simply use **pip**

```
import numpy as np
```
✓ 0.3s Python

Main features

- "The fundamental package for scientific computing with Python"
- Typed multi-dimensional arrays (a.k.a., *matrices*)
- Fast numerical computations
- High-level math functions

What is an NumPy array?

- Homogenous: Elements are all of the same type **`dtype`**
- Array can be indexed
  - By a tuple of nonnegative integers
  - By Booleans
  - By another array
  - By integers
- The **`rank`** of the array is the number of dimensions
- The **`shape`** of the array is a tuple of integers giving the size of the array along each dimension

What is an NumPy array?

```python
a = np.array([1, 2, 3, 4, 5, 6])
print(f"ndim: {a.ndim}; shape: {a.shape}; dtype: {a.dtype}")
```

✓ 0.4s Python

```
ndim: 1; shape: (6,); dtype: int32
```

```python
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]], dtype=np.float64)
print(f"ndim: {a.ndim}; shape: {a.shape}; dtype: {a.dtype}")
print(a)
print(a[0])
```

✓ 0.3s Python

```
ndim: 2; shape: (3, 4); dtype: float64
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]]
[1. 2. 3. 4.]
```


NumPy fundamentals

- Array creation
- Indexing
- Data types
- Broadcasting
- Byte-swapping
- Structured arrays

Array creation: Conversion

- Conversion from other Python structures (i.e., lists and tuples)

```
>>> a1D = np.array([1, 2, 3, 4])
>>> a2D = np.array([[1, 2], [3, 4]])
>>> a3D = np.array([[[1, 2], [3, 4]],
                    [[5, 6], [7, 8]]])
```

- Caveats:
  - dtype (e.g., int8, uint32, float64)

Array creation: Intrinsic

- Intrinsic NumPy array creation functions
  - 1d
    - E.g., `linspace()`, `arange()`
  - 2d
    - E.g., `eye()`, `diag()`, `vander()`
  - Nd
    - E.g., `ones()`, `zeros()`, `full()` , `rand()`, `random()`, `indices()`

Array creation: Intrinsic

- **`np.zeros`**, **`np.ones`**, **`np.full`**: initialized array
  also: **`np.xxx_like`**
- **`np.empty`**: allocates without initializing
- **`np.arange`**: float-friendly range
- **`np.linspace`**: evenly spaced numbers over interval

```
np.linspace(1, 2, 10)
```
✓ 0.7s Python

```
array([1.        , 1.11111111, 1.22222222, 1.33333333, 1.44444444,
       1.55555556, 1.66666667, 1.77777778, 1.88888889, 2.        ])
```


Array creation: Intrinsic

- Random and change shape

```
a = np.random.random(24)
a.shape = (2, 4, 3)
a
```
✓ 0.4s Python

```
array([[[0.09059823, 0.77093022, 0.79452254],
        [0.57293416, 0.5261294 , 0.98325634],
        [0.016108  , 0.60615122, 0.42431647],
        [0.43864826, 0.86614001, 0.38496823]],

       [[0.13047398, 0.45346474, 0.91675888],
        [0.64216519, 0.3850948 , 0.85352943],
        [0.67935171, 0.93377843, 0.32063654],
        [0.61826042, 0.92763883, 0.96025941]]])
```

Shaping

```
a = np.array([1,2,3,4,5,6])
a.shape = (3, 2)
a
```
✓ 0.3s Python

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

```
a.reshape(2, -1)
```
✓ 0.4s Python

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

```
a.ravel()
```
✓ 0.2s Python

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


Array creation: Editing

- Replicating, joining, or mutating existing arrays

```
a = np.ones((4, 1))
b = np.zeros((4, 2))
np.concatenate([a, b], axis=1)
```
✓ 0.4s Python

```
array([[1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.]])
```

Array creation: Editing

- Replicating, joining, or mutating existing arrays

```
A = np.ones((2, 2))
B = np.zeros((2, 4))
C = np.full(shape=(1, 6), fill_value=42)
np.block([[A, B], [C]])
```

✓ 0.9s Python

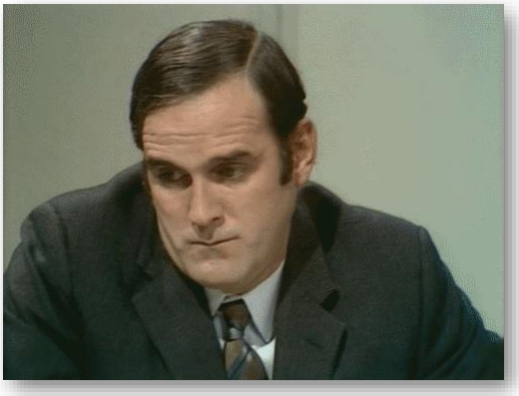
```
array([[ 1.,  1.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  0.,  0.,  0.,  0.],
       [42., 42., 42., 42., 42., 42.]])
```


Array creation: I/O

- Load/save arrays as human-readable text
  - **`loadtxt`**, **`savetxt`**, **`genfromtxt`**
- Load/save arrays as non-human-readable bytes
  - **`load`**, **`save`**, **`savez`**
- … more alternatives to handle blobs, large files, and special library functions (e.g., SciPy, Pandas, OpenCV)

Indexing

- Assignment **vs.** referencing



Indexing: Single element

```
a = np.arange(0, 24, dtype=np.float64)
a.shape = (2, 3, 4)
print(a[1,2,3])
print(a)
print(a[1])
print(a[1][2])
print(a[1][2][3])
```

✓ 0.3s Python

```
23.0
[[[ 0.  1.  2.  3.]
  [ 4.  5.  6.  7.]
  [ 8.  9. 10. 11.]]

 [[12. 13. 14. 15.]
  [16. 17. 18. 19.]
  [20. 21. 22. 23.]]]
[[12. 13. 14. 15.]
 [16. 17. 18. 19.]
 [20. 21. 22. 23.]]
[20. 21. 22. 23.]
23.0
```

Indexing: Slicing

```
a = np.arange(0, 24, dtype=np.float64)
a.shape = (2, 3, 4)
print(a)
print(a[1,:,1:-1])
```

✓ 0.5s Python

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

 [[12. 13. 14. 15.]
  [16. 17. 18. 19.]
  [20. 21. 22. 23.]]]
[[13. 14.]
 [17. 18.]
 [21. 22.]]
```


Indexing: Index arrays

```
a = np.arange(17, 0, -1)
a
```

✓ 0.4s Python

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

```
a[[0, 3, -4, -1]]
```

✓ 0.5s Python

```
array([17, 14,  4,  1])
```

- "Things become more complex when multidimensional arrays are indexed, particularly with multidimensional index arrays"

Indexing: Boolean mask

```
a = np.arange(17, 0, -1)
a % 3 == 0
```

✓ 0.6s Python

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

```
a[a % 3 == 0]
```

✓ 0.1s Python

```
array([15, 12,  9,  6,  3])
```


Indexing: Combining everything

```
a = np.arange(0, 24)
a.shape = (2, 3, 4)
a
```

✓ 0.1s Python

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

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

```
a[0, [2, 0, 1], :]
```

✓ 0.4s Python

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

Indexing: tools

- An `np.newaxis` object can be used within array indices to add new dimensions with a size of 1

Broadcasting

- "Broadcasting" is how NumPy treats arrays with different shapes during arithmetic operations
- The smaller array is "broadcast" across the larger array so that they have compatible shapes

```
a = np.array([18., 5., 23., 10.])
✓ 0.4s                                                    Python

a * 10 + 1
✓ 0.2s                                                    Python
array([181.,  51., 231., 101.])
```

Broadcasting

```
np.zeros((4, 10))
```
✓ 0.2s Python

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

```
np.full(shape=(4,), fill_value=42)
```
✓ 0.3s Python

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

```
np.zeros((4, 10)) + np.full(shape=(10,), fill_value=42)
```
✓ 0.2s Python

```
array([[42., 42., 42., 42., 42., 42., 42., 42., 42., 42.],
       [42., 42., 42., 42., 42., 42., 42., 42., 42., 42.],
       [42., 42., 42., 42., 42., 42., 42., 42., 42., 42.],
       [42., 42., 42., 42., 42., 42., 42., 42., 42., 42.]])
```


Broadcasting

```
a = np.array([18., 5., 23., 10.])
```
✓ 0.4s Python

```
a[:] = 1
a
```
✓ 0.2s Python

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

View vs. Copy

- NumPy functions return either Views or Copies
  - Altering entries of a view, changes the same entries in the original.
- Checkou NumPy documentation
- **np.copy**, **np.view** make explicit copies and views

Data types (partial)

dtype	Corresponding C (C99)
`numpy.bool_`	`bool`
`numpy.byte`	`signed char`
`numpy.int_`	`long`
`numpy.longlong`	`long long`
`numpy.double` `numpy.float64`	`double`
`numpy.longdouble` `numpy.float128`	`long double`
`numpy.cdouble`	`double complex`
`numpy.int8`	`int8_t`
`numpy.int16`	`int16_t`
`numpy.int32`	`int32_t`
`numpy.int64`	`int64_t`

Data types: `astype`

```
a = np.random.rand(4, 4)
a
```
✓ 0.2s Python

```
array([[0.62677893, 0.96942139, 0.56618434, 0.55149847],
       [0.8072098 , 0.58784565, 0.58859556, 0.16784305],
       [0.77261022, 0.52347987, 0.63754229, 0.04488854],
       [0.25669919, 0.05432947, 0.44250618, 0.86913236]])
```

```
(a * 10).astype(int)
```
✓ 0.5s Python

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


Operators

- **`np.transpose`**: permutes axes
- `x.T`: transposes the first 2 axes
- Arithmetic operators are element-wise, in-place operators modify the array
- Logical operators return a bool array
- Universal functions (e.g., **`exp`**, **`sqrt`**, **`sin`**) are element-wise
- Metrix multiplication is `@`

Axes

```
a = np.random.rand(2, 5)
a
```
✓ 0.3s Python

```
array([[0.92101101, 0.47724866, 0.04795136, 0.39730371, 0.41433129],
       [0.0633704 , 0.44462113, 0.05307929, 0.37044329, 0.85022222]])
```

```
a.sum()
```
✓ 0.3s Python

```
4.03958236030496
```

```
a.sum(axis=0)
```
✓ 0.3s Python

```
array([0.98438141, 0.92186979, 0.10103065, 0.76774699, 1.26455351])
```

```
a.sum(axis=1)
```
✓ 0.4s Python

```
array([2.25784603, 1.78173633])
```

```
a.sum(axis=1, keepdims=True)
```
✓ 0.3s Python

```
array([[2.25784603],
       [1.78173633]])
```


Python Data Science

scipy

scipy

- A collection of mathematical algorithms and convenience functions built on the NumPy extension of Python
- **sparse**: Sparse 2-d matrices and associated routines

```
import numpy as np
from scipy import sparse
✓ 0.7s                                                    Python

sparse.lil_matrix((10_000, 10_000))
✓ 0.4s                                                    Python
<10000x10000 sparse matrix of type '<class 'numpy.float64'>'
        with 0 stored elements in List of Lists format>
```


Python Data Science

pandas

pandas

- One of the most popular library that data scientists use
- Labeled axes to avoid misalignment of data
- Created by Wes McKinney in 2008, now maintained by Jeff Reback and many others.
- Key components provided by Pandas:
  - Series
  - DataFrame

Series

- 1-d array-like object of NumPy data

```
import pandas as pd
```
✓ 0.6s Python

```
pd.Series([18, 5, 23, 10])
```
✓ 0.5s Python

```
0    18
1     5
2    23
3    10
dtype: int64
```

Series index

- Both NumPy-like and dict-like behavior

```
s = pd.Series([18, 5, 23, 10], index=['pd', 'pm', 'gd', 'gm'])
```
✓ 0.3s Python

```
s * 2
```
✓ 0.2s Python

```
pd    36
pm    10
gd    46
gm    20
dtype: int64
```

```
s['pm']
```
✓ 0.5s Python

```
5
```


Series index and NaN

```
s = pd.Series([18, 23], index=['Paola', 'Giovanni'])
family = ['Giovanni', 'Paola', 'Gabriele', 'Elena']

pd.Series(s, family)
```
✓ 0.2s Python

```
Giovanni    23.0
Paola       18.0
Gabriele     NaN
Elena        NaN
dtype: float64
```

- Several functions handle NaN and NULL's

Series name and index name

```
s = pd.Series([18, 23], index=['Paola', 'Giovanni'])
family = ['Giovanni', 'Paola', 'Gabriele', 'Elena']


s = pd.Series(s, index=family, name='Squillero\'s')
s.index.name = 'Name'
s
```

✓ 0.3s Python

```
Name
Giovanni    23.0
Paola       18.0
Gabriele     NaN
Elena        NaN
Name: Squillero's, dtype: float64
```


DataFrame

- A tabular data structure of rows and columns (similar to a spreadsheet or database table)
- It may be seen as an order collection of columns
  - Each column can be a different data dtype

```
pd.DataFrame({'name': ['John', 'Paul', 'George', 'Ringo'],
              'surname': ['Lennon', 'McCartney', 'Harrison', 'Starr'] })
```

✓ 0.3s Python

	name	surname
0	John	Lennon
1	Paul	McCartney
2	George	Harrison
3	Ringo	Starr

DataFrame

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                   index=['A', 'B', 'C', 'D', 'E'])
```

✓ 0.7s Python

	year	state	pop	debt
A	2000	Ohio	1.5	NaN
B	2001	Ohio	1.7	NaN
C	2002	Ohio	3.6	NaN
D	2001	Nevada	2.4	NaN
E	2002	Nevada	2.9	NaN

DataFrame

```
pop = {'Nevada': {2001: 2.9, 2002: 2.9}, 'Ohio': {2002: 3.6, 2001: 1.7, 2000: 1.5}}
df = pd.DataFrame(pop)
df
```

✓ 0.1s Python

	Nevada	Ohio
2001	2.9	1.7
2002	2.9	3.6
2000	NaN	1.5

- Tip:
  - Also try `.T`, `.columns`, `.index`, `.name`, `.index.name`

Indexing

- Same philosophy of NumPy, but also supports labels

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
df = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                  index=['A', 'B', 'C', 'D', 'E'])
df[['year', 'pop']]
```

✓ 0.1s Python

	year	pop
A	2000	1.5
B	2001	1.7
C	2002	3.6
D	2001	2.4
E	2002	2.9

Indexing

```
df.state
```

✓ 0.2s Python

```
A      Ohio
B      Ohio
C      Ohio
D    Nevada
E    Nevada
Name: state, dtype: object
```

```
df.loc['A']
```

✓ 0.1s Python

```
year     2000
state    Ohio
pop       1.5
debt      NaN
Name: A, dtype: object
```

Indexing

```
df.loc[['A', 'B']]
```

✓ 0.2s Python

	year	state	pop	debt
A	2000	Ohio	1.5	NaN
B	2001	Ohio	1.7	NaN

```
df.iloc[[0, 1]]
```

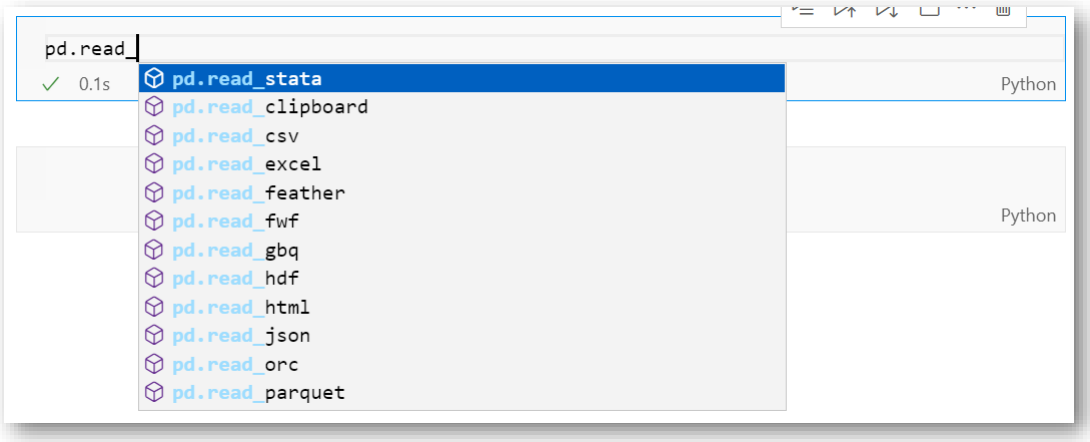
✓ 0.1s Python

	year	state	pop	debt
A	2000	Ohio	1.5	NaN
B	2001	Ohio	1.7	NaN

Modify/Add

- Columns may be modified as *NumPy-like slices*
- Columns may be added **`foo['newcol'] = None`**
- Rows can be modified using **`.loc`** and **`.iloc`**
- Columns may be deleted with **`del`**
- Rows may be deleted with **`.drop`**
- Order of rows/columns may be altered with **`.reindex`**

I/O



Missing Data

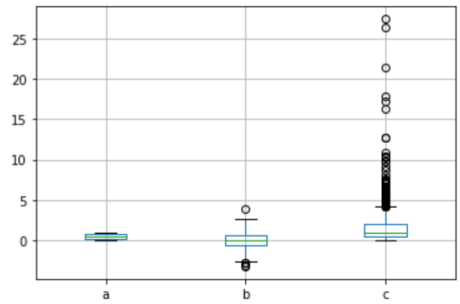
- Drop illegal values with **`dropna()`**, use **`axis=`**
- Replace illegal values with **`fillna()`**
- Replace missing values with **`interpolate()`**, you may needto set **`method='time'`**
- Test with **`notnull()`**

Plot

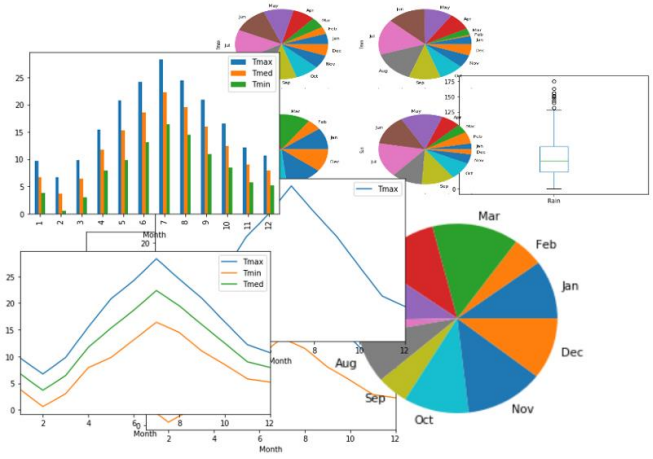
```
df = pd.DataFrame({'a': np.random.rand(1000), 'b': np.random.randn(1000, ),
                   'c': np.random.lognormal(size=(1000,))})
df.boxplot()
```

✓ 0.4s Python

```
<AxesSubplot:>
```



Plot



… also consider using **seaborn**


Google
squillero
squillero
squillero polito
squillero giovanni
squillero github
Google 搜索
手气不错
giovanni.squillero@polito.it