

Scientific Programming in Python

Inteligencia Artificial en los Sistemas de Control Autónomo
Máster Universitario en Ingeniería Industrial

Departamento de Automática

Objectives

1. Introduce some Python tools for scientific programming.
2. Motivate the need of efficient matrix manipulation.
3. Handle matrices and dataframes in Python.
4. Basic data visualization with Python.

Bibliography

Jake VanderPlas. Python Data Science Handbook. Chapters 1, 2, 3 and 4. O'Reilly. (Link).

Table of Contents

1. Overview

- Data Science
- The data scientist toolkit
- Anaconda
- Python IDEs for Data Science

2. Basics

- Magic commands
- Pasting code blocks
- Running external code
- Input and output history
- iPython shell commands
- Automagic

3. NumPy

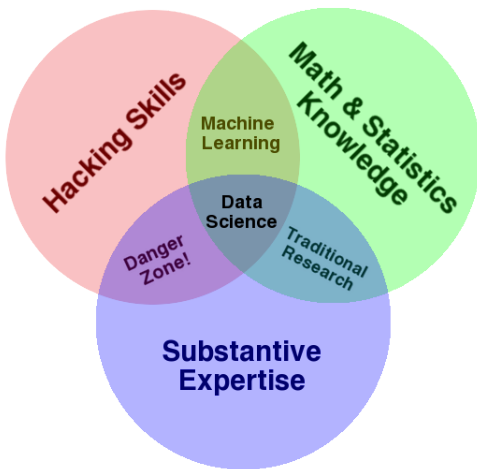
- Understanding Data Types in Python
- Introduction
- Matrix creation
- NumPy data types
- NumPy array attributes
- Accessing single elements

- Accessing subarrays
- Reshaping of arrays
- Concatenation of arrays
- Splitting of arrays
- Universal functions
- Aggregations
- Broadcasting
- Structured arrays

4. Pandas

- Introduction
- The Pandas **Series** object
- The Pandas **DataFrame** object
- Constructing **DataFrame** objects
- Data indexing and selection
- Operating on data
- Missing data
- Combining datasets: `pd.concat()`
- Combining datasets: `pd.merge()`
- Aggregation in Pandas
- Grouping in Pandas

Data Science



Data Science

The data scientist toolkit (I)

Data science is about manipulating data

- Need of specialized tools
- Two main languages: R and Python

Python is a general purpose programming language

- Easy integration
- Huge ecosystem of packages and tools

Need of data-oriented tools

- Features provided by third-party tools

Data Science

The data scientist toolkit (II)

Tool	Type	Description
iPython	Software	Advanced Python interpreter
Jupyter	Software	Python notebooks (Python interpreter)
Numpy	Package	Efficient array operations
Pandas	Package	Dataframe support
Matplotlib	Package	Data visualization
Seaborn	Package	Data visualization with dataframes
Scikit-learn	Package	AI/ML package for Python

Data Science

Anaconda

All those tools are packaged in Anaconda

- Python distribution for Data Science

Anaconda provides Spyder

- Python IDE designed for Data Science

Other tools provided by Anaconda

- Conda: Packages management tool
- TensorFlow: Deep Learning
- Many others



Data Science

Python IDEs for Data Science (I)

iPython

iPython = Interactive
Python

- Extended functionality
- Enhanced UI
- External editor

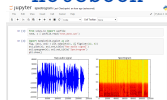
Running iPython:
\$ ipython

Jupyter

Python notebooks

- Web-based IDE
- Documentation
- Integration with GitHub
- Uses iPython

Running Jupyter:
\$ jupyter
notebook



Rodeo

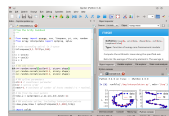
Python version of RStudio

- Good for R developers
- Not included in Anaconda
- Uses iPython



Spyder

Matlab-like IDE



Data Science

Python IDEs for Data Science (II)

Exercises

Write a Python script that shows the multiplication table of the number 5. Write the script using each one of the following environments:

1. iPython + text editor of your choice.
2. Jupyter.
 - Bonus track: Publish the notebook in GitHub.
3. Spyder.
4. Rodeo.

iPython

Basics (I)

In regular Python ...

- most objects come with a docstring attribute
- docstring accesible thorough `help()`

iPython provides `?`, a shortcut to `help()`

- `len?`, `list?`, `list.append?`
- Try to type just `'?'`

Easy access to source code with `??`

- Does not work with most builtin functions!

iPython

Basics (II)

Press <tab> to complete almost everything

- Object contents

```
In [21]: a = [1,2,3]
In [22]: a.
```

- Packages

```
In [26]: import num
```

- Wildcards

```
In [29]: *Warning?
%%!
ArithmeticError      BaseException
BlockingIOError      BlockingIOError
AssertionError        BrokenPipeError
AttributeError         BufferError
```

iPython

Basics (III): Keyboard shortcuts

Navigation

KEYSTROKE	ACTION
Ctrl-a	Move cursor to the beginning of the line
Ctrl-e	Move cursor to the end of the line
Ctrl-b	Move cursor back one character
Ctrl-f	Move cursor forward one character

History

KEYSTROKE	ACTION
Ctrl-p (↑)	Previous command
Ctrl-n (↓)	Next command
Ctrl-r	Reverse-search

Text entry

KEYSTROKE	ACTION
Ctrl-d	Delete next character in line
Ctrl-k	Cut text from cursor to end of line
Ctrl-u	Cut text from beginning of line to cursor
Ctrl-y	Yank (paste) previously cut text

iPython

iPython magic commands

Magic commands: iPython extension of Python syntax

- Not valid in regular Python
- Provides handy features
- Widely used in DS and ML

Two flavours

- % prefix: Line magics - single line
- %% prefix: Cell magics - several lines

Help available

- %magic: Magic commands
- %lsmagic: List of magic commands

iPython

Input and output history (II)

Fast access to history: Underscore ()

- Variable containing the last output
- Example: `print(_)`

Double and triple underscores

- Example: `print(__)`
- Example: `print(____)`

Trick: Shortcut to access `(_n)`

- Out[n] = _n, with n=number
- Example: `print(_2)`

Magic command to show history

- %history

Supressing command output (;)

- Example: $4 * 2$;

iPython

iPython shell commands

iPython provides easy interaction with the shell

- Execution of shell commands from iPython
- Use prefix `!`
- Example: `!ls`, `!pwd`

Save shell output in Python variables

- Example: `files = !ls`

Use Python variables in shell

- Example: `!echo {files}`

iPython
Automagic

Problems with some shell commands

```
In [23]: !pwd
/repositorios/pythonCourse
In [24]: !cd ..
In [25]: !pwd
/repositorios/pythonCourse
```

Some magic commands here to help

- %cd,%ls,%mkdir,%pwd,
- ...

Those magics are regularly used ...

- ... so common that % is no longer required (automagic)
- Working with iPython is almost like working with a Unix-like shell

Automagic commands

cat, cp, env, ls, man, mkdir, more,
mb, pwd, rm and rmdir

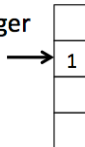
Understanding Data Types in Python (II)

Dynamic typing must be implemented somewhere ...

Python 3.4 source code

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

C Integer



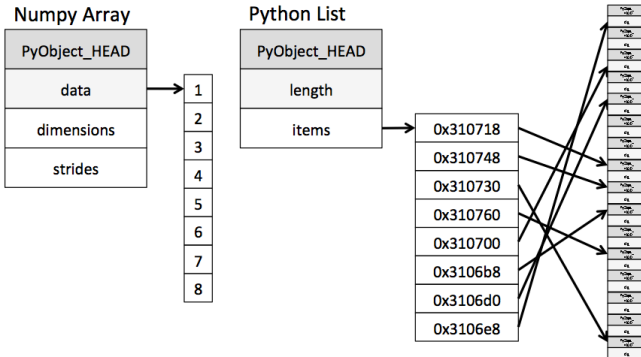
Python Integer



Understanding Data Types in Python (III)

A Python list may contain different types

```
In [1]: L3 = [True, "2", 3.0, 4]
...: [type(item) for item in L3]
Out[1]: [bool, str, float, int]
```



NumPy

Understanding Data Types in Python (IV)

Standard Python data types are powerful and flexible

- Flexibility has a price: Reduced performance
- Not an big issue in generic programming
- A big issue in scientific programming
- We require efficient data manipulation mechanisms: NumPy

NumPy: Python package for numeric computation

- Efficient array implementation
- Fast mathematical functions
- Random numbers generation
- Static data types: Less flexibility

Most Python modules for AI/ML depend on NumPy, in particular

- Pandas (dataframes), Scikit-learn (ML), Seaborn (data visualization)

NumPy

Introduction

NumPy must be imported in order to be available

- Remember, you can use `np?` or `np.<TAB>`

The main component of NumPy is `ndarray`

- Python object
- Efficient matrix representation
- Homogeneous elements

Convention

```
import numpy as np
```

```
In [1]: array = np.array
        ([1, 2, 3])
In [2]: array
Out[1]: array([1, 2, 3])
In [3]: array = np.array
        ([[1, 2], [3, 4]])
```


NumPy

Accessing single elements

Unidimensional array

- `array[index]`

Unidimensional array from the end

- `array[-index]`

Multidimensional array

- `array[row,column]`

```
x = np.array([5, 0, 3, 3, 7, 9])
x[0] # 5
x[4] # 7
x[-1] # 9
x[-2] # 7
x = np.array([[3, 5, 2, 4],
              [7, 6, 8, 8],
              [1, 6, 7, 7]])
x[2, 0] # 1
x[2, -1] # 7
```

Warning

Ndarray has fixed types, values can be truncated without warning. Big source of problems!

NumPy

Reshaping of arrays

Reshaping arrays is a very common task

- Change data number of dimensions

Important ndarray method: `reshape()`

- Changes the dimensions of an array
- Sizes must match

Conversion of 1-D arrays into column or row matrices

- Using method `reshape()`
- Using the keyword `np.newaxis`

General reshaping

```
In [1]: x=np.array([1, 2, 3, 4])
In [2]: x.reshape((2,2))
Out [1]:
array([[1, 2],
       [3, 4]])
```

1-D to row

```
x = np.array([1, 2, 3])
x.reshape((1, 3))
x[np.newaxis, :]
```

1-D to column

```
x.reshape((3, 1))
x[:, np.newaxis]
```

NumPy

Concatenation of arrays

Three methods to join arrays

- `np.concatenate()`
- `np.vstack()`
- `np.hstack()`

`np.concatenate()`

```
In [1]: x = np.array([1, 2, 3])
In [2]: y = np.array([3, 2, 1])
In [3]: np.concatenate([x, y])
Out[1]: array([1, 2, 3, 3, 2, 1])
```

`np.vstack()`

```
In [1]: x = np.array([1, 2, 3])
In [2]: grid = np.array([[9, 8, 7],
...                      [6, 5, 4]])
In [3]: np.vstack([x, grid])
Out[107]:
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
```

NumPy

Splitting of arrays

Three methods to split arrays

- `np.split()`
- `np.vsplit()`
- `np.hsplit()`

`np.split()`

```
In [1]: x = [1, 2, 3, 99, 99, 3, 2, 1]
In [2]: x1, x2, x3 = np.split(x, [3, 5])
In [3]: print(x1, x2, x3)
[1 2 3] [99 99] [3 2 1]
```

`np.vstack()`

```
In [1]: grid = np.arange(16).reshape((4, 4))
In [2]: print(grid)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
In [3]: upper, lower = np.vsplit(grid, [2])
In [4]: print(upper)
[[0 1 2 3]
 [4 5 6 7]]
```


NumPy

Universal functions (I)

Python may be ridiculously slow

- Run-time type checks and function dispatching
- Evident when an operation is repeated over a collection of data

Performance test

```
def compute_reciprocals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1.0 / values[i]
    return output

big_array = np.random.randint(1, 100, size=1000000)
# Standard CPython
%timeit compute_reciprocals(big_array)
# 3.59 s ± 139 ms per loop
# NumPy
%timeit (1.0 / big_array)
# 5.41 ms ± 182 µs per loop
```

NumPy

Universal functions (II)

Vectorized operations: Functions that are aware of NumPy's static typing

- Avoid dynamic type-checking
- Loop related code pushed into the compiled layer
- Hugely improved performance
- Perform an operation with the first element and then it to the rest

In NumPy, vectoriced operations are named **universal functions**, of **ufuncs**

- Regular functions
- Arrays as arguments (one or multi-dimensional)
- Operates between arrays of different sizes (**broadcasting**)

In order to take advantage of NumPy's performance, ufuncs must be used

NumPy

Universal functions: Arithmetic functions

NumPy makes use of Python's native arithmetic operators

- Used like regular Python operators
- Operators are wrappers for NumPy's functions

OPERATOR	EQUIVALENT UFUNC	DESCRIPTION
+	<code>np.add</code>	Addition (e.g., $1 + 1 = 2$)
-	<code>np.subtract</code>	Subtraction (e.g., $3 - 2 = 1$)
-	<code>np.negative</code>	Unary negation (e.g., -2)
*	<code>np.multiply</code>	Multiplication (e.g., $2 * 3 = 6$)
/	<code>np.divide</code>	Division (e.g., $3 / 2 = 1.5$)
//	<code>np.floor_divide</code>	Floor division (e.g., $3 // 2 = 1$)
**	<code>np.power</code>	Exponentiation (e.g., $2 ** 3 = 8$)
%	<code>np.mod</code>	Modulus/remainder (e.g., $9 \% 4 = 1$)

NumPy

Universal functions (III)

Binary ufuncs

```
x = np.arange(4)
print("x      =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
print("x / 2 =", x / 2)
print("x // 2 =", x // 2) # floor division
np.add(x, 2)             # array plus scalar
```

Unary ufuncs

```
print("-x      =", -x)
print("x ** 2 =", x ** 2)
print("x % 2  =", x % 2)
```

NumPy

Universal functions: Basic functions

Absolute value

- `np.absolute(x)` and `np.abs(x)`

Trigonometric functions

- `np.sin(theta)`, `np.cos(theta)`, `np.tan(theta)`
- `np.arcsin(theta)`, `np.arccos(theta)`, `np.arctan(theta)`

Exponents and logarithms

- `np.exp(x)`, `np.exp2(x)`, `np.power(base, x)`
- `np.log(x)`, `np.log2(x)`, `np.log10(x)`

Advanced mathematical functions

- Checkout module `scipy.special` for exotic mathematical functions

Output as argument

- Avoid temporal variables using `out` argument in ufuncs
- Example: `np.multiply(x, 10, out=y)`

NumPy

Universal functions: Special functions

Aggregation functions

- Applied to any ufunc
- `reduce(x)`: Repeatedly applies an ufunc to the elements of an array until only a single result remains
- `accumulate(x)`: Like `reduce()`, but it stores intermediate values
- `outer(x)`: Compute the output of all pairs of two different inputs

reduce() example

```
In [1]: x = np.arange(1, 6)
In [2]: np.add.reduce(x)
Out [1]: 15
```

accumulate() example

```
In [1]: np.add.reduce(x)
Out [1]: 15
```

Outer() example

```
In [132]: np.multiply.outer(x, x)
array([[ 1,  2,  3,  4,  5],
       [ 2,  4,  6,  8, 10],
       [ 3,  6,  9, 12, 15],
       [ 4,  8, 12, 16, 20],
       [ 5, 10, 15, 20, 25]])
```

NumPy

Universal functions: Aggregations (I)

Many ufuncs to summarize data

- Basic step in exploratory data analysis
- Argument `axis` determines to which dimension the summary is to be applied

FUNCTION	NaN-SAFE VERSION	DESCRIPTION
<code>np.sum</code>	<code>np.nansum</code>	Compute sum of elements
<code>np.prod</code>	<code>np.nanprod</code>	Compute product of elements
<code>np.mean</code>	<code>np.nanmean</code>	Compute mean of elements
<code>np.std</code>	<code>np.nanstd</code>	Compute standard deviation
<code>np.var</code>	<code>np.nanvar</code>	Compute standard deviation
<code>np.min</code>	<code>np.nanmin</code>	Find minimum value
<code>np.max</code>	<code>np.nanmax</code>	Find maximum value
<code>np.argmin</code>	<code>np.nanargmin</code>	Find index of minimum value
<code>np.argmax</code>	<code>np.nanargmax</code>	Find index of maximum value
<code>np.median</code>	<code>np.nanmedian</code>	Compute median of elements
<code>np.percentile</code>	<code>np.nanpercentile</code>	Compute rank-based statistics of elements
<code>np.any</code>	N/A	Evaluate whether any elements are true
<code>np.all</code>	N/A	Evaluate whether all elements are true

NumPy

Universal functions: Aggregations (II)

(Download dataset)

- Use `wget` or `curl` to download the file within `iPython`

Basic data analysis example

```
import pandas as pd
data = pd.read_csv('president_heights.csv')
heights = np.array(data['height(cm)'])
print(heights)

print("Mean height:      ", heights.mean())
print("Standard deviation:", heights.std())
print("Minimum height:   ", heights.min())
print("Maximum height:   ", heights.max())

print("25th percentile:  ", np.percentile(heights, 25))
print("Median:           ", np.median(heights))
print("75th percentile:   ", np.percentile(heights, 75))
```

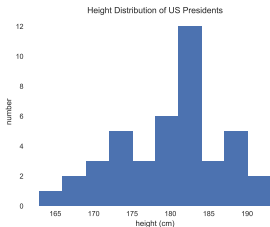

NumPy

Universal functions: Aggregations (III)

Basic data analysis example (Continuation)

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # set plot style

plt.hist(heights)
plt.title('Height Distribution of US Presidents')
plt.xlabel('height (cm)')
plt.ylabel('number');
```



NumPy

Universal functions: Broadcasting (I)

Broadcasting is a mechanism to operate over arrays of different sizes

- Used in ufuncs
- Implicit array expansion through three rules

Broadcasting rules

1. Rule 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. Rule 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. Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

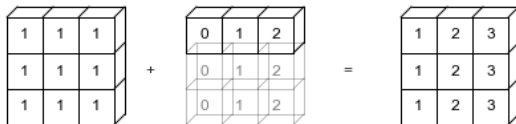
NumPy

Universal functions: Broadcasting (II)

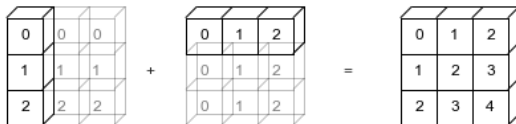
`np.arange(3)+5`



`np.ones((3,3))+np.arange(3)`



`np.arange(3).reshape((3,1))+np.arange(3)`



Array expansion does not
consume memory!

NumPy

Universal functions: Broadcasting (III)

Normalization

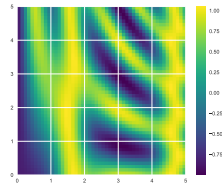
```
X = np.random.random((10, 3))
Xmean = X.mean(0)
X_centered = X - Xmean
```

3D plot

```
%matplotlib inline
import matplotlib.pyplot as plt

x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 50)[: , np.newaxis]
z = np.sin(x)**10 + np.cos(10 + y*x)*np.cos(x)

plt.imshow(z, origin='lower',
           extent=[0, 5, 0, 5], cmap='viridis')
plt.colorbar();
```



NumPy

Comparisons, masks, and Boolean logic (I)

(Download dataset)

Example

```
import numpy as np
import pandas as pd

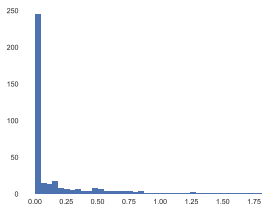
# pandas to extract rainfall inches as a ndarray
rainfall = pd.read_csv('Seattle2014.csv')['PRCP'].values
inches = rainfall / 254.0 # 1/10mm -> inches
inches.shape
# Outputs (365,)
```



```
%matplotlib
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
plt.hist(inches, 40);
```

NumPy

Comparisons, masks, and Boolean logic (II)



Data filtering is a recurrent task

- How many rainy days were there in the year?
- What is the average precipitation on those rainy days?
- How many days were there with more than half an inch of rain?

Two filtering methods in NumPy

- Boolean arrays masks
- Fancy indexing

NumPy

Comparisons, masks, and Boolean logic: Booleans arrays masks (I)

Syntax examples

```
x [ x < 5 ]
x [ x == 3 ]
x [ ( x > 3 ) & ( x <= 5 ) ]
```

We've seen arithmetic ufuncs ...

- ... but they also support comparison and boolean operations
- Return an array of booleans

OPERATOR	UFUNC
<code>==</code>	<code>np.equal</code>
<code>!=</code>	<code>np.not_equal</code>
<code><</code>	<code>np.less</code>
<code><=</code>	<code>np.less_equal</code>
<code>></code>	<code>np.greater</code>
<code>>=</code>	<code>np.greater_equal</code>

OPERATOR	UFUNC
<code>&</code>	<code>np.bitwise_and</code>
<code> </code>	<code>np.bitwise_or</code>
<code>^</code>	<code>np.bitwise_xor</code>
<code>~</code>	<code>np.bitwise_not</code>

NumPy

Comparisons, masks, and Boolean logic: Booleans arrays masks (II)

Example

```
print(x)
[[5, 0, 3, 3]
 [7, 9, 3, 5]
 [2, 4, 7, 6]]
np.count_nonzero(x < 6) # Returns 8
np.sum(x < 6) # Returns 8
np.sum(x < 6, axis=1) # By row, returns
    array([4, 2, 2])
np.any(x > 8) # Returns True
np.any(x < 0) # Returns False
np.all(x < 10) # Returns True

np.sum(~((inches <= 5) | (inches >= 1)))
```


NumPy

Comparisons, masks, and Boolean logic: Fancy indexing

So far we've seen three accessing methods

- Simple indices (`x[1]`)
- Slices (`x[:5]`)
- Boolean masks (`x[x>0]`)

Fancy indexing: Pass arrays on indices instead of scalars

```
x = rand.randint(100, size=10)
[x[3], x[7], x[2]] # Simple indices
ind = [3, 7, 4] # Array of indices
x[ind] # Fancy indexing
x[[3, 5, 6]] # Also valid
```

The shape of the result reflects the shape of the index arrays rather than the shape of the array being indexed

NumPy

Structured arrays (I)

Some times, we need to group data

- Example: Store name, age and weight of several people
- Different data types for each attribute

```
name = [ 'Alice ', 'Bob ', 'Cathy ', 'Doug ' ]
age = [ 25, 45, 37, 19 ]
weight = [ 55.0, 85.5, 68.0, 61.5 ]
```

Solution: Structured arrays

```
# Use a compound data type for structured arrays  
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),  
                          'formats':('U10', 'i4', 'f8')})
```

NumPy

Structured arrays (II)

```
data[ 'name' ] = name
data[ 'age' ] = age
data[ 'weight' ] = weight

# Get all names
data[ 'name' ]

# Get first row of data
data[0]

# Get the name from the last row
data[ -1 ][ 'name' ]

# Get names where age is under 30
data[ data[ 'age' ] < 30 ][ 'name' ]
```

These kind of structures are day-to-day used

- Pandas is a much better choice

Pandas

Introduction

A data science workflow needs more features

- Label columns and rows
- Missing data
- Operations on groups
- Data input

Pandas implements all those features, and more

- Built on NumPy's ndarray

Pandas provides two main objects

- Series
- DataFrame

Convention

```
import numpy as np
import pandas as pd
```

Pandas

Introduction

A DS/ML workflow needs more features

- Missing data
- Data input
- Operations on groups
- Label columns and rows

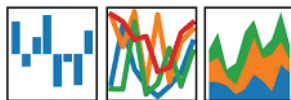
Pandas provides all those features, and more

- Pandas = **P**ANel **D**Ata **S**ystem
- Built on NumPy's ndarray
- Provides **dataframes**

Pandas provides two main objects

- Series and DataFrame

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$


Convention

```
import numpy as np
import pandas as pd
```

Pandas

The Pandas Series object (I)

A **Series** is a one-dimensional array of indexed data

- NumPy arrays indices are implicit (i.e. its position)
- Series indices are explicit, and can be any type

INDEX	VALUES
'a'	0.25
'b'	0.5
'c'	0.75
'd'	0.99

Two attributes

- `values`: ndarray
- `index`: `pd.Index` object

Two indices

- Implicit: Regular index
- Explicit: Custom index

```
data = pd.Series([0.25,
                  0.5, 0.75, 1.0])
data.values
data.index
data[1:3]
```

Pandas

The Pandas Series object (II)

```
In [1] : data = pd.Series([0.25, 0.5, 0.75, 1.0],
                           index=['a', 'b', 'c', 'd'])
```

```
In [2]: data
```

Out [1] :

a 0.25

b 0.50

c 0.75

d 1.00

```
dtype: float64
```

```
In [3]: data['a']
```

Out [2]: 0.25

```
In [4]: data[o]
```

Out [3]: 0.25

Pandas

The Pandas DataFrame object (II)

```
In [1]: import seaborn as sns
```

```
In [2]: iris = sns.load_dataset('iris')
```

```
In [3]: iris.head()
```

Out [1] :

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

```
In [246]: iris.columns
```

Out [2 4 6]:

```
Index(['sepal_length', 'sepal_width', 'petal_length',  
      'petal_width', 'species'], dtype='object')
```

Pandas

Constructing DataFrame objects

Manual initialization

- From a single Series object
`pd.DataFrame(population, columns=['population'])`
- From several Series objects
`pd.DataFrame('population': population, 'area': area)`
- From a dictionary
`pd.DataFrame([{'a': 0, 'b': 0}, {'a': 1, 'b': 2}])`
- From a NumPy 2-D array
`pd.DataFrame(np.random.rand(3, 2),
columns=['foo', 'bar'], index=['a', 'b', 'c'])`

Read from a file

- CSV (very common!!!): `pd.read_csv('filename.csv')`
- Excel:
`pd.read_excel('filename.xlsx', sheetname='mysheet')`

Pandas

Data indexing and selection: Series

Dictionary-like syntax

```
>>> data = pd.Series([0.25, 0.5,
                      0.75, 1.0], index=['a', 'b',
                      'c', 'd'])
```

```
>>> 'a' in data
True
```

```
>>> data.keys()
Index(['a', 'b', 'c'], dtype='object')
```

```
>>> list(data.items())
[('a', 0.25), ('b', 0.5), ('c', 0.75)]
```

```
>>> data[ 'e' ] = 1.25
```

Array-like syntax

```
>> data[ 'a' : 'c' ] #Explicit index
```

a 0.25

b 0.50

c 0.75

```
dtype: float64
```

```
>> data[0:2] # Implicit index
```

a 0.25

b 0.50

```
dtype: float64
```

```
>> data[data > 0.5] # Masking
```

c 0.75

d 1.00

```
dtype: float64
```

```
>> data[['b', 'c']] # Fancy index
```

b 0.50

c 0.75

```
dtype: float64
```

Pandas

Data indexing and selection: DataFrame

Dictionary-like syntax

```
>>> data['area']
>>> data.area
>>> data.area is data['area']
True
>>> data['density'] = data['pop']
    / data['area']
```

Array-like syntax

```
>>> data.values # Get values
array
>>> data.T # Transpose
>>> data[0] # First row
>>> data['area'] # Area column
```

Remember indexing conventions

- Indexing refers to columns (`data['area']`)
- Slicing refers to rows (`data['Florida':'Illinois']`)
- Masking refers to rows (`data[data.density > 100]`)

Pandas

Data indexing and selection: loc, iloc and ix

Two types of indices in Pandas

- Explicit and implicit
- Indexing (`data[0]`) is explicit
- Slicing (`data[:2]`) is implicit (Python-like)
- Source of troubles!

Pandas makes explicit the used scheme

- loc: Explicit index
- iloc: Implicit index
- ix: Hybrid

Series

```
>>> serie.loc[1]
```

```
>>> serie.loc[1:3]
```

```
>>> serie.iloc[1]
```

```
>>> serie.iloc[1:3]
```

Dataframes

```
>>> df.iloc[:3, :2]
```

```
>>> df.loc[:, 'illinois', : 'pop']
```

```
>>> df.ix[:3, : 'pop']
```

```
>>> df.loc[df.data > 100, ['pop',
                             'density']]
```

```
>>> df.iloc[0, 2] = 90
```

Pandas

Operating on data (I)

Pandas fully supports NumPy's ufuncs

- Efficient computations

Additional Pandas features

- Index and column name preservation
- Index aligning
- Easy data combination

```
>>> rng = np.random.RandomState(42)
>>> df = pd.DataFrame(rng.randint(0,
    10, (3, 4)))
>>> df = pd.DataFrame(rng.randint(0,
    10, (3, 4)), columns=[ 'A', 'B', 'C',
    , 'D' ])
>>> print(df)
   A  B  C  D
0  7  2  5  4
1  1  7  5  1
2  4  0  9  5
>>> np.sin(df * np.pi / 4)
      A      B      C      D
0 -7.07e-01  1.0 -0.7  1.22e-16
1  7.07e-01 -0.7 -0.7  7.07e-01
2  1.22e-16  0.0  0.7 -7.07e-01
```


Pandas

Combining datasets: `pd.concat()` (I)

Many times we need to combine two or more datasets

- Pandas provides `pd.concat()`, `append()` and `pd.merge()`

pd.concat() signature

```
pd.concat(objs, axis=0, join='outer',
          join_axes=None, ignore_index=False, keys
          =None, levels=None, names=None,
          verify_integrity=False, copy=True)
```

By default, `pd.concat()` joins rows preserving index

- `axis`: Join columns (`axis=1`)
- `verify_integrity`: Raise error if duplicates (`verify_integrity=True`)
- `ignore_index`: Create new index (`ignore_index=True`)
- `join`: Can be 'outer' (union) or 'inner' (intersection)

Pandas

Combining datasets: `pd.concat()` (II)

```
>> df1 = pd.DataFrame([{'A': 'Ao', 'B': 'Bo'}, {'A': 'A1', 'B': 'B1'}
    ])
>> df2 = pd.DataFrame([{'A': 'A2', 'B': 'B2'}, {'A': 'A3', 'B': 'B3'}
    ])

>> print(df1), print(df2); print(pd.concat([df1, df2]))
  A  B      A  B      A  B
0 Ao Bo    0 A2 B2  0 Ao Bo
1 A1 B1    1 A3 B3  1 A1 B1
                   0 A2 B2
                   1 A3 B3

>> pd.concat([df1, df2], axis=1)
  A  B  A  B
0 Ao Bo A2 B2
1 A1 B1 A3 B3

>> df1.append(df2)
```

Pandas

Combining datasets: `pd.merge()` (I)

Merging based on relational algebra

- Similar to databases tables joins
- Pretty intelligent figuring out the desired output
- By default, join dataframes using shared columns names

Pandas

Combining datasets: `pd.merge()` (II)

One-to-one

```
>> print(df1); print(df2)
employee      group
0      Bob  Accounting
1      Jake  Engineering
2      Lisa  Engineering
3      Sue      HR
employee  hire_date
0      Lisa      2004
1      Bob      2008
2      Jake      2012
3      Sue      2014
>> print(pd.merge(df1, df2))
employee  group  hire_date
0      Bob  Accounting  2008
1      Jake  Engineering  2012
2      Lisa  Engineering  2004
3      Sue   HR          2014
```

Many-to-one

```
>>> print(df3); print(df4)
  employee  group  hire_date
0      Bob  Accounting  2008
1      Jake  Engineering  2012
2      Lisa  Engineering  2004
3       Sue           HR  2014

  group  supervisor
0  Accounting  Carly
1  Engineering  Guido
2           HR  Steve

>>> print(pd.merge(df3, df4))
  employee  group  hire_date  supervisor
0      Bob  Accounting  2008      Carly
1      Jake  Engineering  2012      Guido
2      Lisa  Engineering  2004      Guido
3       Sue           HR  2014      Steve
```

Pandas

Combining datasets: `pd.merge()` (III)

Many-to-many

```
>>> print(df1); print(df5)
```

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

	group	skills
0	Accounting	math
1	Accounting	spreadsheets
2	Engineering	coding
3	Engineering	linux
4	HR	spreadsheets
5	HR	organization

```
>>> pd.merge(df1, df5)
```

	employee	group	skills
0	Bob	Accounting	math
1	Bob	Accounting	spreadsheets
2	Jake	Engineering	coding
3	Jake	Engineering	linux
4	Lisa	Engineering	coding
5	Lisa	Engineering	linux
6	Sue	HR	spreadsheets
7	Sue	HR	organization

Pandas

Combining datasets: `pd.merge()` (IV)

pd.merge() signature

```
pd.merge(left, right, how='inner', on=None,
         left_on=None, right_on=None, left_index=
         False, right_index=False, sort=False,
         suffixes=('_x', '_y'), copy=True,
         indicator=False, validate=None)
```

Arguments:

- `on`: Key column name
- `left_on`: Left table key column name
- `right_on`: Right table key column name
- `how`: Set arithmetic, `'inner'` (default, intersection), `'outer'` (union, fills missings with NaNs), `'left'` (left entries), `'right'` (right entries)

Pandas

Combining datasets: `pd.merge()` (V)

```
>>> A
   lkey  value
0  foo    1
1  bar    2
2  baz    3
3  foo    4

>>> B
   rkey  value
0  foo    5
1  bar    6
2  qux    7
3  bar    8

>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
   lkey  value_x  rkey  value_y
0  foo    1      foo    5
1  foo    4      foo    5
2  bar    2      bar    6
3  bar    2      bar    8
4  baz    3      NaN    NaN
5  NaN    NaN     qux    7
```



```
>>> import seaborn as sns
>>> planets = sns.load_dataset('planets')
>>> planets.head()
```

		method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006	
1	Radial Velocity	1	874.774	2.21	56.95	2008	
2	Radial Velocity	1	763.000	2.60	19.84	2011	
3	Radial Velocity	1	326.030	19.40	110.62	2007	
4	Radial Velocity	1	516.220	10.50	119.47	2009	

```
>>> planets.dropna().describe()
```

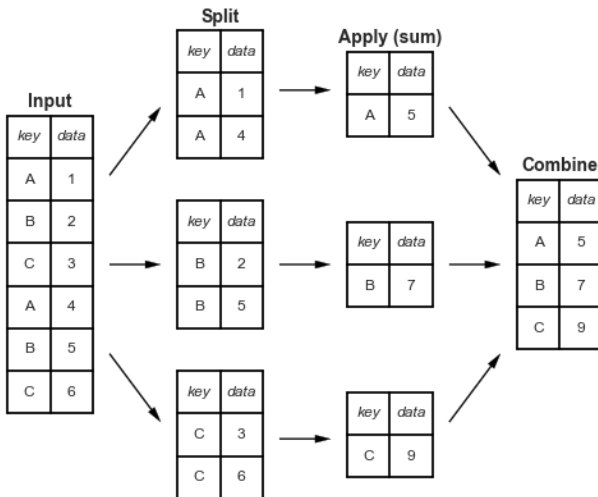
	number	orbital_period	mass	distance	year
count	498.00	498.000000	498.00	498.0000	498.000
mean	1.73	835.778671	2.50	52.0682	2007.377
std	1.17	1469.128259	3.63	46.5960	4.167
min	1.00	1.328300	0.00	1.3500	1989.000
25%	1.00	38.272250	0.21	24.4975	2005.000
50%	1.00	357.000000	1.24	39.9400	2009.000
75%	2.00	999.600000	2.86	59.3325	2011.000
max	6.00	17337.500000	25.00	354.0000	2014.000

```
>>> planets.mean()
```

number	1.785507
orbital_period	2002.917596
mass	2.638161
distance	264.069282
year	2009.070531
dtype:	float64

Pandas

Grouping in Pandas (II)



Pandas

Grouping in Pandas (III)

C 7