

# **MIE1624H – Introduction to Data Science and Analytics**

## **Lecture 2 - Python Programming**

# Roadmap

## **Python essentials**

Variables and types

Operators and comparisons

Compound types - strings, tuples, lists and dictionaries

Functions

Modules

Files and the operating system

Control flow - conditional statements (if, elif, else), loops

Exception handling

## **Introduction to Pandas**

Introduction to pandas data structures – DataFrame, index objects

Pandas essential functionality

Summarizing and computing descriptive statistics

Pivot tables in pandas

IPython notebooks

# Introducing Python

Python is an interpreted language (not a compiled one)

=> you can run code incrementally, one statement at a time

# **Expressions and Values**

# Arithmetic Operators

Operator	Operation	Expression	English description	Result
+	addition	11 + 56	11 plus 56	67
-	subtraction	23 - 52	23 minus 52	-29
*	multiplication	4 * 5	4 multiplied by 5	20
**	exponentiation	2 ** 5	2 to the power of 5	32
/	division	9 / 2	9 divided by 2	4.5
//	integer division	9 // 2	9 divided by 2	4
%	modulo (remainder)	9 % 2	9 mod 2	1

# Arithmetic Operator Precedence - 1

When multiple operators are combined in a single expression, the operations are evaluated in order of precedence.

Operator	Precedence
<b>**</b>	<b>highest</b>
<b>- (negation)</b>	
<b>*, /, //, %</b>	
<b>+ (addition), - (subtraction)</b>	<b>lowest</b>

# Arithmetic Operator Precedence - 2

```
>>> 10 % 3
```

```
1
```

```
>>> 25 + 30 / 6
```

```
30.0
```

```
>>> 5 + 4 ** 2
```

```
21
```

```
>>> 100 - 25 * 3 % 4
```

```
97
```

```
>>> 3 + 2 + 1 - 5 + 4 % 2 - 1 / 4 + 6
```

```
6.75
```

# Types

A ***type*** is a set of *values* and the *operations* that can be performed on those values.



# Types `int` and `float`

`int`: integer

3, 4, 894, 0, -3, -18

`float`: floating point number (an approximation to a real number)\

5.6, 7.342, 53452.0, 0.0, -89.34, -9.5

```
>>> 5 + 2 * 4
13
```

```
>>> 5.0 + 2 * 4
13.0
```

```
>>> 30 / 6
5.0
```

```
>>> 30 // 6
5
```

# Type **str** - 1

- A *string literal* is a sequence of characters. A string can be made up of letters, numbers, and special characters.
- Strings in Python start and end with a single quote (') or double quotes (").
- If a string begins with a single quote, it must end with a single quote. The same applies to double-quoted strings.
- **The two types of quotes cannot be mixed!**

# Type **str** - 2

```
>>> "Hello!"
```

```
'Hello!'
```

```
>>> 'how are you?'
```

```
'how are you?'
```

```
>>> 'short- and long-term'
```

```
'short- and long-term'
```

# Dictionaries -1

- Lists/Tuples/Strings are **ordered** collections
- Dictionaries are **mapped** (unordered) collections
  - data is accessed using keys
  - data is unordered

- Example:

*my\_dictionary = {key\_1:data\_1, key\_2:data\_2,... , key\_n:data\_n}*

*key\_1* maps to *data\_1*, *key\_2* maps to *data\_2*, etc.

*my\_dict = {'Name': 'Jane Smith', 'SN': 990632802, 'Status': 'Registered'}*

# Dictionaries - Properties

- A dictionary keeps track of associations between keys and values
  - like a regular dictionary where a **key** is a word and the **value** is the definition of that word.
    - ▶ This makes look-ups and other operations very easy
- Dictionary values can be any Python object ( standard objects or user-defined objects.
- Dictionary keys must be immutable objects: strings, numbers, tuples, etc.:
  - no duplicate key is allowed

# Dictionary Functions

- *cmp(dict1, dict2)*      Compares elements of both dict
- *len(dict)*      Gives the total length of the dictionary,  
i.e., the number of items in the dictionary
- *str(dict)*  
dictionary      Produces a string representation of a

# Dictionary Methods

- **clear()** Removes all elements of the dictionary
- **copy()** Returns a shallow copy of the dictionary
- **fromkeys()** Create a new dictionary with keys from seq and values set to value.
- **get(key, default=None)** Returns the value key maps to or default if key is not in the dictionary
- **has\_key(key)** Returns true if key is in the dictionary, false otherwise
- **items()** Returns the data in the dictionary as a list of tuples (key, value)
- **keys()** Returns the keys in the dictionary as a list
- **update(dict2)** Adds dict2's key-values pairs
- **values()** Returns the values in the dictionary as a list

- Form to define a dictionary:

```
my_dict = {key1: value1, key2: value2, ...}
```

- Form to look up a key's value:

```
my_dict[key]
```

- Form to add or update a key-value pair:

```
my_dict[key] = value
```

- Form to delete from a dictionary

```
del dict['Name']    # remove entry with key 'Name'
```

```
dict.clear()        # remove all entries in dict
```

```
del dict             # delete entire dictionary
```

- `my_dict` is the name of the dictionary object.

- `key` is any immutable object (string, int, tuple).

- `value` is any Python object.



# Example -1

```
>>> CO2_by_year = {1799:1, 1800:70, 1801:74,  
... 1802:82, 1902:215630, 2002:1733297}
```

```
>>> # Look up the emissions for the given year
```

```
>>> CO2_by_year[1801]  
74
```

```
>>> # Add another year to the dictionary
```

```
>>> CO2_by_year[1950] = 734914
```

```
>>> CO2_by_year{1799: 1, 1800: 70, 1801: 74,  
... 1802: 82, 1902: 215630, 2002: 1733297,  
... 1950: 734914}
```

# Example -2

```
>>> CO2_by_year[2009] = 1000000
```

```
>>> CO2_by_year[2000] = 10
```

```
>>> CO2_by_year
```

```
{2000: 10, 2002: 1733297, 1799: 1, 1800: 70, 1801: 74, 1802: 82, 2009: 1000000, 1902: 215630}
```

```
>>> 1950 in CO2_by_year
```

```
False
```

```
>>> del CO2_by_year[1950]
```

```
>>> len(CO2_by_year)
```

```
8
```

```
>>> for key in CO2_by_year:
```

```
...     print(key)
```

```
2000
```

```
2002
```

```
...
```

- How can we iterate through the keys?

```
for k in d.keys():  
    print(k)
```

- How can we iterate through the values?

```
for v in d.values():  
    print(v)
```

- How can we iterate through the key-value pairs?

```
for key, value in dict.items():
```

# Tuples

- Are similar to lists, but cannot be changed

```
my_tuple = (1, 2, 3)
```

- Useful when we want to group data together, in assignment statements

# Sets

- Like mathematical sets, they are unordered!

```
>>> my_set = {1, 2, 3}
```

```
>>> x = [1, 2, 3]
```

```
>>> my_set = set(x)
```

- An element is either in the set or it is not.  
→ efficient membership check

# Assignment statements - 1

- General form of an assignment statement: **variable = expression**
- General rule for executing an assignment statement:
  1. Evaluate the expression to the right of the = sign.
    - produces a memory address of the value the expression evaluates to
  2. Store the memory address in the variable on the left of the = sign.

# Assignment statements - 2

```
>>> difference = 20
```

```
>>> double = 2 * difference
```

```
>>> double
```

```
40
```

```
>>> difference = 5
```

```
>>> double
```

```
40
```

- The expression on the right of the = sign is evaluated to 20
- The value 20 will be put at memory address **id1**.
- The variable on the left of the = sign, `difference`, will refer to 20 by storing **id1** in `difference`.

# Assignment statements - 3

```
>>> double = 2 * difference
```

- The expression on the right of the = sign: `2 * difference` is evaluated
  - => `difference` refers to the value 20
  - => `difference * 20` evaluates to 40.
- The memory address `id2` is assigned to the value 40.
- The variable on the left of the = sign, `double`, will refer to 40 by storing `id2`.



# Assignment statements - 4

```
>>> base = 20
>>> height = 12
>>> area = base * height / 2
>>> area
120.0
>>> celsius = 22
>>> fahrenheit = celsius * 9/5 + 32
>>> fahrenheit
71.6
```

# Assignment statements - 5

**Strings can also be stored as variables.**

```
>>> reminder_text = 'Buy groceries after work'
>>> reminder_text
'Buy groceries after work'
>>> str_var = 'Welcome to APS106'
>>> str_var
'Welcome to APS106'
>>> str_var = "What is 10 * (2 + 9)?"
>>> str_var
'What is 10 * (2 + 9)?'
```

# Assignment statements - 6

**Strings can also be stored as variables.**

```
>>> reminder_text = 'Please buy groceries after work'
```

```
>>> reminder_text
```

```
'Please buy groceries after work'
```

```
>>> str_var = 'Welcome to APS106'
```

```
>>> str_var
```

```
'Welcome to APS106'
```

```
>>>str_var = "What is 10 * (2 + 9)?"
```

```
>>> str_var
```

```
'What is 10 * (2 + 9)?'
```

# Augmented Assignment Operators

```
>>> number = 3
```

```
>>> number
```

```
3
```

```
>>> number = 2 * number
```

```
>>> number
```

```
6
```

```
>>> number = number * number
```

```
>>> number
```

```
36
```

```
>>> score = 50>>> score50>>> score = score + 20
```

```
>>> score70
```

# Augmented Assignment Operators

Operator	Expression	Identical Expression	English description
<b>+=</b>	<b>x = 7</b> <b>x += 2</b>	<b>x = 7</b> <b>x = x + 2</b>	<b>x refers to 9</b>
<b>-=</b>	<b>x = 7</b> <b>x -= 2</b>	<b>x = 7</b> <b>x = x - 2</b>	<b>x refers to 5</b>
<b>*=</b>	<b>x = 7</b> <b>x *= 2</b>	<b>x = 7</b> <b>x = x * 2</b>	<b>x refers to 14</b>
<b>/=</b>	<b>x = 7</b> <b>x /= 2</b>	<b>x = 7</b> <b>x = x / 2</b>	<b>x refers to 3.5</b>
<b>//=</b>	<b>x = 7</b> <b>x //= 2</b>	<b>x = 7</b> <b>x = x // 2</b>	<b>x refers to 3</b>
<b>%=</b>	<b>x = 7</b> <b>x %= 2</b>	<b>x = 7</b> <b>x = x % 2</b>	<b>x refers to 1</b>
<b>**=</b>	<b>x = 7</b> <b>x **= 2</b>	<b>x = 7</b> <b>x = x ** 2</b>	<b>x refers to 49</b>

# What is a function?

- A function is a block of organized (**reusable**) code that is used to perform an activity.
- In Python a function is implemented as a *compound statement*.
- Python has **built-in** functions, but programmers can also create their own **user-defined** functions.

# Defining a Python Function -1

- The general form of a function definition:

```
def function_name (parameters):  
    '''function_docstring'''  
    function_body  
    [return [expression]]
```

- A **function block** begins with the keyword **def** followed by the function name and parentheses ( ).
- **Parameters/arguments:**
  - 0 or more, separated by a comma, are placed within the parentheses.
  - variables whose values are supplied when the function is called
  - by default, parameters have a positional behaviour (exception: named arguments, which can be given in any order)

# Defining a Python Function -2

**Function body:** consists of one or more statements,

- The code block/body within every function starts with a colon (:) and is *indented*.
- The first statement of a function can be an optional statement - the documentation string of the function, a.k.a. the docstring.
- The statement **return[expression]** exits if a function is passing back a value to the caller.
  - => A return statement with no arguments is the same as **return None**.



# Using Functions

- Defining a function only gives the function a name, declares its input parameters and specifies its behaviour, i.e., the instructions to be performed when the function is executed => **but nothing gets executed yet!**
- Once the definition of a function is ready, the function can be executed by **calling** it from another function or directly from the Python prompt.

# Calling a Function

- The general form of a function call:

**`function_name (arguments )`**

- Executing a function call:
  - > Evaluate the arguments
  - > Call the function, passing in the argument values
    - >> the instructions in the body of the function are carried out

# Function Design Recipe - Six Steps

1. Pick a meaningful name: a short answer to 'What does the function do'?
2. Prepare the Type Contract and write the function header
  - > What are the parameter types?
  - > What type of value is returned?
  - *Pick meaningful parameter names: it is much easier to understand a function if the variables have names that reflect their meaning.*
3. Prepare a few examples (function calls) - 'What should the function do'?
4. Description: write a docstring describing the function. Mention every parameter in your description. Describe the return value.
5. Body - Write the body of the function.
6. Test - Run the examples designed in Step 3 to make sure they work as expected.

# Applying the Design Recipe -1

The United States measures temperature in Fahrenheit and Canada measures it in Celsius. When travelling between the two countries it helps to have a conversion function. Write a function that converts from Fahrenheit to Celsius.

1. **Pick a name:** `convert_to_celsius`

2. **Type Contract and Header** (what the function will look like)\

Type Contract\ `(number) -> number`

Header\ `def convert_to_celsius(fahrenheit):`

3. **Examples**

`convert_to_celsius(32)\ => 0`

`convert_to_celsius(212)\ => 100\`

4. **Description**\- Return the number of Celsius degrees equivalent to fahrenheit degrees.

5. **Body**

```
degrees = (fahrenheit - 32) * 5 / 9
return degrees
```

# Applying the Design Recipe -2

## Complete function definition

```
def convert_to_celsius(fahrenheit):  
    ''' (number) -> number  
    Return the celsius degrees equivalent to  
    fahrenheit degrees.  
    '''  
  
    celsius = (fahrenheit - 32) * 5 / 9  
  
    return celsius
```

## 6. Test - run the examples.

```
>>> convert_to_celsius(32)  
0  
>>> convert_to_celsius(212)  
100
```

# Calling functions within other function definitions -1

Let us write a function to convert from hours to seconds.

```
def convert_to_minutes(num_hours):  
    """(number) -> number  
    Return the number of minutes there are in num_hours  
    hours.  
    """  
    result = num_hours * 60  
    return result  
  
def convert_to_seconds(num_hours):  
    """(number) -> number  
    Return the number of seconds there are in num_hours  
    hours.  
    """  
    return convert_to_minutes(num_hours) * 60
```

Testing:

```
>>> convert_to_minutes(2)\  
120\  
>>> convert_to_seconds(2)\  
7200\
```

# Calling functions within other function definitions -2

```
def convert_to_celsius(fahrenheit):  
    ''' (number) -> number  
    Return the number of celsius degrees  
    equivalent to fahrenheit degrees.  
    '''  
  
    degrees = (fahrenheit - 32) * 5 / 9  
    return degrees  
  
def convert_to_kelvin(fahrenheit):  
    ''' (number) -> number  
    Return the number of kelvin degrees equivalent  
    to fahrenheit degrees.  
    '''  
  
    kelvin = convert_to_celsius(fahrenheit) + 273.15  
    return kelvin
```

## Testing:

```
>>> convert_to_kelvin(32)  
273.15
```

# Use Function Calls as Arguments to Other Functions

One triangle has a base of length 3.8 and a height of length 7.0 and a second triangle has a base of length 3.5 and a height of length 6.8. Find the area of the larger triangle.

The approach: pass calls to function area as arguments to built-in function max.

```
>>> max(triangle_area(3.8, 7.0), triangle_area(3.5, 6.8))
```



# Modules

- A module is a file containing Python definitions and statements.
- The file name is the module name with the suffix .py appended.
- Example: fibo.py module

# Fibonacci numbers module

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

- When needed just: `import fibo`

# Opening Files

- `open(filename, mode)`

(str,str) -> io.TextIOWrapper

opens the file Filename

in the same directory as the .py file

returns a file-handle

mode can take several values:

r: open the file for reading

w: open the file for writing (erasing the content!)

a: open the file for writing, appending new  
information to the end of the file

# Opening Files -2

- To start using a file, given its filename, it has to be open. (The name is a string.)
- To open the file, use the function `open()`

```
myfile = open("story.txt", "r")
```

- `open()` is a Python function
- `story.txt` is the name of the file to be open
- `myfile` is a variable that is assigned the file object returned by `open`
- `r` is a string indicating what we wish to do with the file.

Options for this string are "r", "w", "a", meaning read, write or append. The default is "r"

**Note:** writing to a file that already exists, erases the existing content. Use append if you want to preserve the content.

# Closing Files

```
myfile.close()
```

→ (NoneType) -> NoneType

→ `myfile` is the file object returned by `open()`

# Reading Files

- We call a file object that was opened for reading a **reader**
- Various ways to read from a reader:

1. Read lines one at a time from beginning to end:

for line in myf le:

<statements>

2. Read everything in the file at once into a list of strings: read the whole file into list str\_ls. Each element of str\_ls is a line.

```
str_ls = myf le.readlines()
```

```
print(str_ls)
```

3. Read everything in the file at once into a string:

```
s = myfile.read() # Read the whole file into string s.  
print(s)
```

4. Read a certain number of characters:

```
s = myfile.read(10) # Read 10 characters into s  
print(s)
```

5. Read a line at a time:

```
s = myfile.readline() # Read a line into s.  
print(s)  
  
s = myfile.readline() # Read the next line into s.  
print(s)
```

# Reading Files - Recap

- `myfile.readline()` - read 1 line from the file
- `myfile.read()` - read the whole file into a single string
- `myfile.readlines()` - read the whole file into a list, with each element being one line of text
- `myfile.readlines(n)` - read the next N bytes of a file, rounded up to the end of a line.

# Reading CSV in Python

```
import csv

...

input_file = open(file_name)
reader = csv.reader(input_file)

...

for line in reader:

    # Read as from an ordinary file, but line
    is a list

    <process line>
```



# Writing to a file

- First we open a file to write, then we write the contents

**filehandle.write()**

Just like printing, except you have to add your own newline characters

- Close your file

# Introduction to Pandas

- an open source Python library providing high performance data structures and analysis tools.

```
>>> import pandas as pd
```

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

```
>>> import matplotlib.pyplot as plt
```

# Pandas Data Structures -Series

- One-dimensional labeled array
- Holds any data type (integers, strings, floating point numbers, Python objects, etc.)
- The axis labels are collectively referred to as the index.

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

- **data:** a dictionary, an ndarray, a scalar value (e.g., 11)
- **index:** is a list of axis labels.

# Series from from an ndarray

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

```
>>> s  
a      0.2735  
b      0.6052  
c     -0.1692  
d      1.8298  
e      0.5432  
dtype: float64
```

```
>>> s.index  
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

```
>>> pd.Series(np.random.randn(5))  
0      0.3674  
1     -0.8230  
2     -1.0295  
3     -1.0523  
4     -0.8502  
dtype: float64
```

# Series from from a dictionary

- If an **index** is passed, the values in data corresponding to the labels in the index will be pulled out.
- If no **index** is passed, an index will be constructed from the sorted keys of the dict, if possible.

```
>>> d = {'a' : 0., 'b' : 1., 'c' : 2.}
```

```
>>> pd.Series(d)
```

```
a    0.0
```

```
b    1.0
```

```
c    2.0
```

```
dtype: float64
```

```
>>> pd.Series(d, index=['b', 'c', 'd', 'a'])
```

```
b    1.0
```

```
c    2.0
```

```
d    NaN
```

```
a    0.0
```

```
dtype: float64
```

- **NOTE:** NaN is the standard missing data marker used in pandas

# Series from from a scalar value

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

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

# Series Behaviour

- Series acts very similarly to a ndarray, and is a valid argument to most NumPy functions.

```
>>> s[0]
>>> 0.27348116325673794
>>> s[:3]
>>> a      0.2735
      b      0.6052
      c     -0.1692
      dtype: float64
```

- A Series is like a fixed-size dict in that you can get and set values by index label:

```
>>> s['a']
>>> 0.27348116325673794
>>> s['e'] = 12.

>>> s.get('a')
>>> 0.27348116325673794
```

# DataFrame Objects

- `class pandas.DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)[source]`
- Two-dimensional, size-mutable, potentially heterogeneous tabular data structure with labeled rows and columns.
- Dictionar-like container for Series objects.
- Arithmetic operations align on both row and column labels.



# DataFrame - Parameters

- **data** : numpy ndarray, dictionary (Series, arrays, constants, or list-like objects), or DataFrame
- **index** : index or array-like to use for resulting frame.
- **columns** : Index or array-like, labels to use for resulting frame.
- **dtype** : data\_type (default None), to force, otherwise infer
- **copy** : boolean (default False), to copy data from inputs.

```
>>> d = {'col1': ts1, 'col2': ts2}
```

```
>>> df1 = DataFrame(data = d, index = index)
```

```
>>> df2 = DataFrame(numpy.random.randn(10, 5))
```

```
>>> df3 = DataFrame(numpy.random.randn(10, 5),  
columns=['a', 'b', 'c', 'd', 'e'])
```

- `numpy.random.randn` returns a sample(s) from the "standard normal" distribution.

# DataFrames from Series or dictionaries

- The result index will be the union of the indexes of the various Series.

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

```
>>> df = pd.DataFrame(d)
```

```
>>> df
```

```
one two  
a  1.0  1.0  
b  2.0  2.0  
c  3.0  3.0  
d  NaN  4.0
```

```
>>> pd.DataFrame(d, index=['d', 'b', 'a'])
```

```
one two  
d  NaN  4.0  
b  2.0  2.0  
a  1.0  1.0
```

# Accessing Rows and Columns

- The row and column labels can be accessed, respectively, by accessing the index and columns attributes:
- **Note:** when a particular set of columns is passed along with a dict of data, the passed columns override the keys in the dict.

```
>>> df.index
```

```
Index([u'a', u'b', u'c', u'd'], dtype='object')
```

```
>>> df.columns
```

```
Index([u'one', u'two'], dtype='object')
```

# Index

- Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects
- Parameters:
- data : array-like (1-dimensional)
- dtype : NumPy dtype (default: object)
- copy : bool Make a copy of input ndarray
- name : objectName to be stored in the index
- tupleize\_cols : bool (default: True)

When True, attempt to create a MultiIndex if possible

# Index Attributes

## Attributes

<b>T</b>	return the transpose, which is by definition self
<b>asi8</b>	
<b>base</b>	return the base object if the memory of the underlying data is
<b>data</b>	return the data pointer of the underlying data
<b>dtype</b>	
<b>dtype_str</b>	
<b>flags</b>	
<b>has_duplicates</b>	
<b>hasnans</b>	
<b>inferred_type</b>	
<b>is_all_dates</b>	
<b>is_monotonic</b>	alias for is_monotonic_increasing (deprecated)
<b>is_monotonic_decreasing</b>	return if the index is monotonic decreasing (only equal or
<b>is_monotonic_increasing</b>	return if the index is monotonic increasing (only equal or
<b>is_unique</b>	
<b>itemsize</b>	return the size of the dtype of the item of the underlying data

# Index Methods

## Methods

<code>all(*args, **kwargs)</code>	Return whether all elements are True
<code>any(*args, **kwargs)</code>	Return whether any element is True
<code>append(other)</code>	Append a collection of Index options together
<code>argmax([axis])</code>	return a ndarray of the maximum argument indexer
<code>argmin([axis])</code>	return a ndarray of the minimum argument indexer
<code>argsort(*args, **kwargs)</code>	Returns the indices that would sort the index and its underlying data.
<code>asof(label)</code>	For a sorted index, return the most recent label up to and including the passed label.
<code>asof_locs(where, mask)</code>	where : array of timestamps
<code>astype(dtype[, copy])</code>	Create an Index with values cast to dtypes.
<code>copy([name, deep, dtype])</code>	Make a copy of this object.
<code>delete(loc)</code>	Make new Index with passed location(-s) deleted
<code>difference(other)</code>	Return a new Index with elements from the index that are not in <i>other</i> .
<code>drop(labels[, errors])</code>	Make new Index with passed list of labels deleted
<code>drop_duplicates(*args, **kwargs)</code>	Return Index with duplicate values removed

# Reshaping by pivoting DataFrame objects -1

- Reshaping by pivoting DataFrame objects
- Data is often stored in CSV files or databases in so-called “stacked” or “record” format:

```
>>> df
```

	date	variable	value
0	2000-01-03	A	0.469112
1	2000-01-04	A	-0.282863
2	2000-01-05	A	-1.509059
3	2000-01-03	B	-1.135632
4	2000-01-04	B	1.212112
5	2000-01-05	B	-0.173215
6	2000-01-03	C	0.119209
7	2000-01-04	C	-1.044236
8	2000-01-05	C	-0.861849
9	2000-01-03	D	-2.104569
10	2000-01-04	D	-0.494929
11	2000-01-05	D	1.071804

# Reshaping by pivoting DataFrame objects -2

- To select out everything for variable A we could do:

```
>>> df[df['variable'] == 'A']  
>>> date          variable    value  
0 2000-01-03      A          0.469112  
1 2000-01-04      A          -0.282863  
2 2000-01-05      A          -1.509059
```

- For time series operations a better representation would have the columns as unique variables and an index of dates identifying individual observations.
- To reshape the data use the **pivot** function:

```
>>> df.pivot(index='date', columns='variable', values='value')  
>>> variable          A          B          C          D  
date  
2000-01-03    0.469112   -1.135632    0.119209   -2.104569  
2000-01-04   -0.282863    1.212112   -1.044236   -0.494929  
2000-01-05   -1.509059   -0.173215   -0.861849    1.071804
```



# Computing Descriptive Statistics

- `DataFrame.describe(percentiles=None, include=None, exclude=None)[source]`
- Generate various summary statistics, excluding NaN values.
- **Parameters:**
- **percentiles** : array-like, optional. The percentiles to include in the output. Should all be in the interval [0, 1].
- **include, exclude** : list-like, 'all', or None (default) Specify the form of the returned result. Either:
  - None to both (default). The result will include only numeric-typed columns or, if none are, only categorical columns.
  - A list of dtypes or strings to be included/excluded.
  - If **include**= 'all', the output column-set will match the input one.

**Returns:** summary statistics

# Jupyter Notebook

- An *interactive computational environment*, in which you can combine code execution, rich text, mathematics, plots and multi media.
- *Notebook documents* ( “notebooks”) are documents produced by the Jupyter Notebook App  
contain: computer code (e.g. python) and rich text elements (paragraph, equations, figures, links, etc.).
- Notebook documents are both human-readable documents as well as executable documents which can be run to perform data analysis.

# Jupyter Notebook App

- A server-client application that allows editing and running notebook documents via a web browser.
- The Jupyter Notebook App can be executed on a local desktop requiring no internet access or can be installed on a remote server and accessed through the internet.
- In addition to displaying/editing/running notebook documents, the App has a “Dashboard” showing local files and allowing to open notebook documents.

# Notebook Kernel

*A computational engine* that executes the code contained in a Notebook document.

- The kernel associated with a notebook is automatically launched when the notebook is opened.
- When the notebook is executed, the kernel performs the computation and produces the results.

# Notebook Dashboard

- The Dashboard is the component which is shown first in the Jupyter App.
- Main functionality: open notebook documents, and to manage the running kernels.
- Other features (similar to a file manager): navigating folders and renaming/deleting files.

# Running the Jupyter Notebook

- The App can be launched by clicking on the Jupyter Notebook icon installed by Anaconda in the start menu (Windows) or by typing in a terminal (cmd on Windows, Terminal on OSX):  
  
    > jupyter notebook
- This will launch a new browser window/tab showing the Notebook Dashboard
- When started, the App can access only files within its start-up folder (including any sub-folder).
- If the notebook documents are not in a subfolder of your user folder further configuration steps are necessary.

# Shutting Down/Restarting a Kernel

- When a notebook is opened, its kernel is also started.
- Closing the notebook browser tab, will not shut down the kernel => the kernel needs to be explicitly shut down.
- To **shut down** a kernel:
  - (1) go to the associated notebook and click on menu **File ->Close and Halt**.
  - or (2) in the **Running** tab of the **Dashboard** (which shows all the running notebooks/kernels) click on a the **Shutdown** button of the kernel you want to stop.
- To **restart** a kernel click on the menu **Kernel -> Restart**.  
This can be useful to start over a computation from scratch

# Executing a Notebook

- Launch the App
- In the Dashboard, navigate to find the notebook.
- Click on its name (will open it in a new browser tab).
- Run the notebook step-by-step (one cell a time) by pressing **shift + enter**.
- Run a notebook in a single step by clicking on the menu **Cell -> Run All**.



# Shut down the Jupyter Notebook App

- Closing the browser window/tab will not close the Jupyter Notebook App.  
=> to completely shut it down the associated terminal needs to be closed.
- Many copies of the Jupyter Notebook App can be run in parallel, but it is not a recommended usage mode.