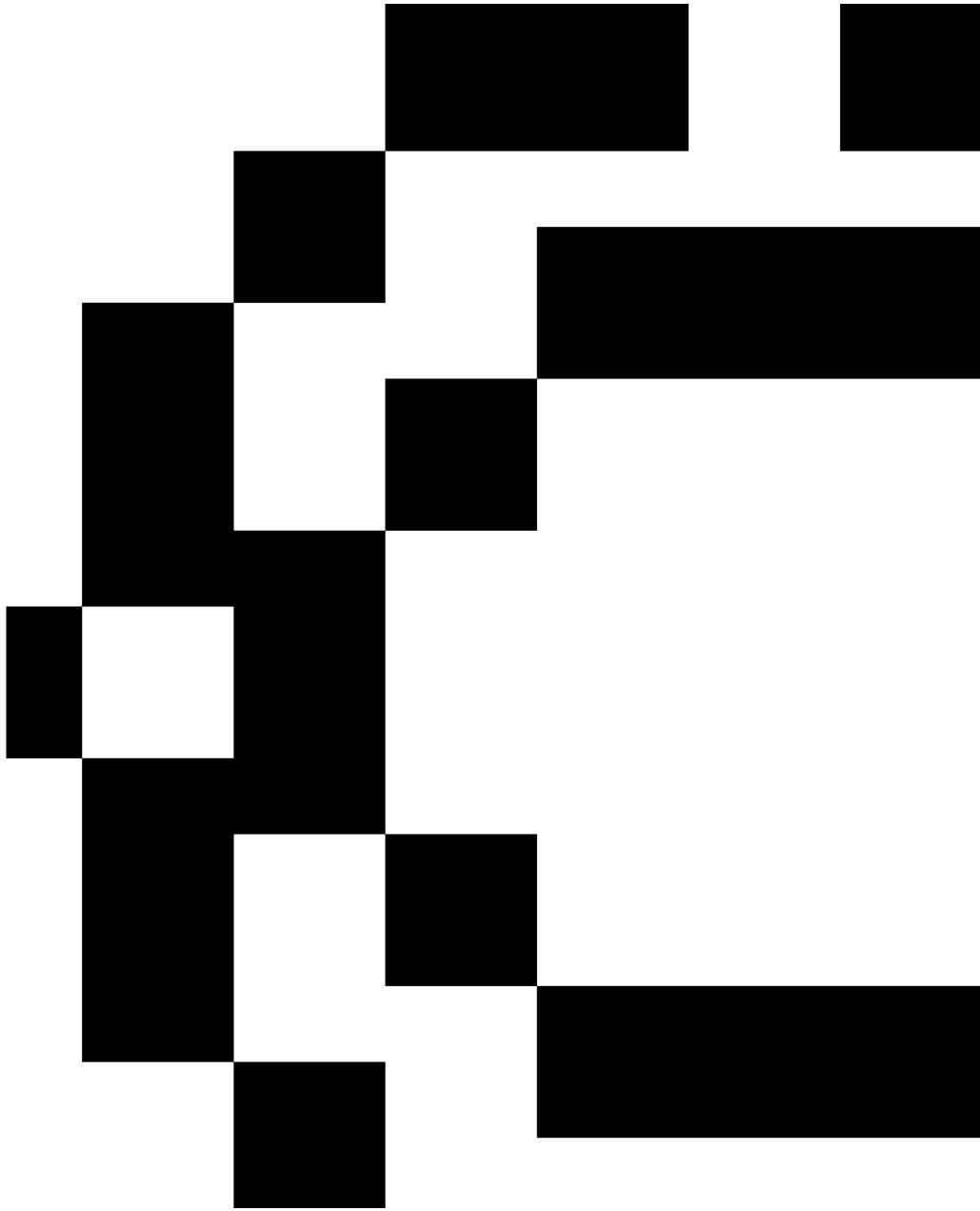
NOVA IMS INFORMATION MANAGEMENT SCHOOL

Programming for Data Science

Lecture 4

Flávio L. Pinheiro

fpinheiro@novaims.unl.pt | www.flaviolpp.com
www.linkedin.com/in/flaviolpp | X @flavio_lpp



Quizzes

https://www.socrative.com

Login as a student

Room Name: **PDS2025**

Student ID: **Student Number**

or

https://api.socrative.com/rc/5NpKRX

Student ID: **Student Number**

Please Join Now!

Point of Situation

Variables and Data Structures

```
lista = [1,2,3,4,5,6,7,8,9,10]
for i,value in enumerate(lista):
    if value == 0 or value == 1:
        lista[i] = 1
    else:
        temp = value
        lista[i] = 1
        while temp > 0:
            lista[i] *= temp
            temp -= 1
```

```
print(lista)
```

```
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Loops & Functions

```
lista = [1,2,3,4,5,6,7,8,9,10]
for i,value in enumerate(lista):
    if value == 0 or value == 1:
        lista[i] = 1
    else:
        temp = value
        lista[i] = 1
        while temp > 0:
            lista[i] *= temp
            temp -= 1
```

takes a list of values and replaces each value with the factorial. If the value is equal to 1 or 0 it replaces it with 1.

```
print(lista)
```

```
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

```
lista = [5,3,2,1,0]
print(enumerate(lista))
```

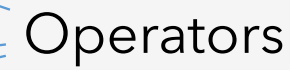
```
<enumerate object at 0x103cf9630>
```

```
print(list(enumerate(lista)))
```

```
[(0, 5), (1, 3), (2, 2), (3, 1), (4, 0)]
```

Operators

```
lista = [1,2,3,4,5,6,7,8,9,10]
for i,value in enumerate(lista):
    if value == 0 or value == 1:
        lista[i] = 1
    else:
        temp = value
        lista[i] = 1
        while temp > 0:
            lista[i] *= temp
            temp -= 1
```




```
print(lista)
```

```
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Indentations are Important!

```
lista = [1,2,3,4,5,6,7,8,9,10]
for i,value in enumerate(lista):
    if value == 0 or value == 1:
        lista[i] = 1
    else:
        temp = value
        lista[i] = 1
        while temp > 0:
            lista[i] *= temp
            temp -= 1
```

For Loop

If and Else Statement

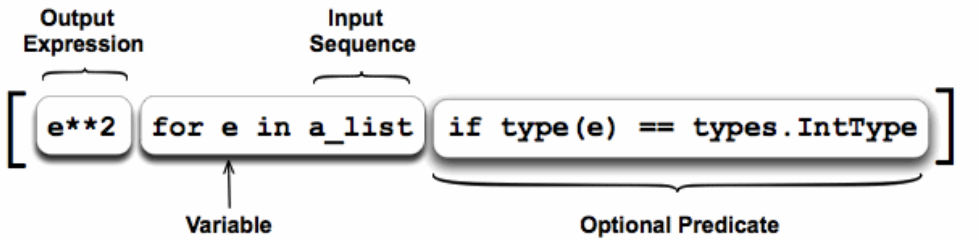
While Loop

```
print(lista)
```

```
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Comprehensions

List comprehensions




```
lista = [i for i in range(1,10)]
print(lista)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

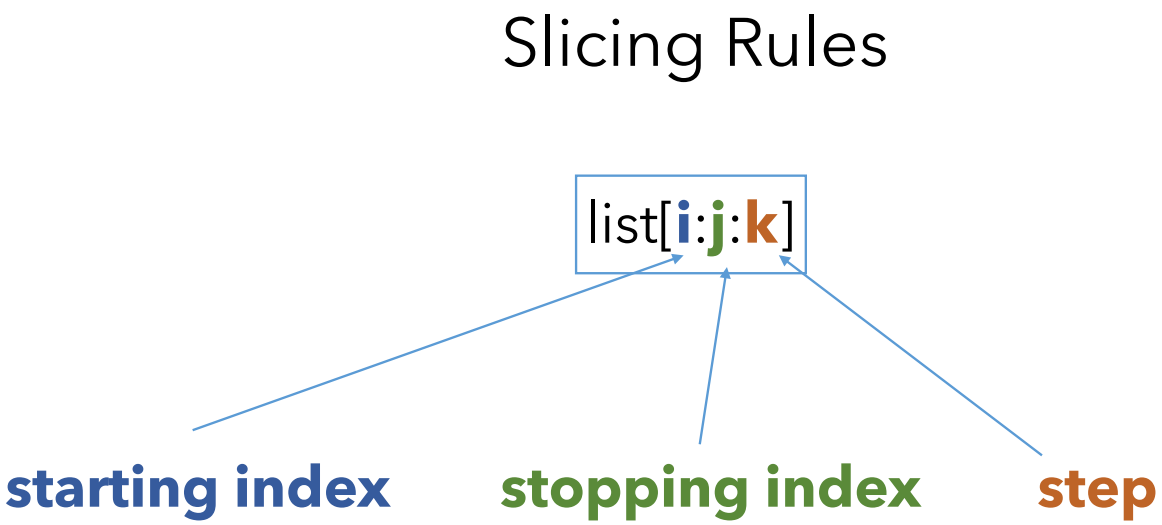
```
lista = [i**2 for i in range(1,10)]
print(lista)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
lista = [i for i in range(1,10) if i%2 == 0]
print(lista)
```

```
[2, 4, 6, 8]
```

Slices



What we will not Covered Here

in the readings that might want to take a look at

- Objects (Classes, etc)
- iterators
- Decorators
- Generators
- Handling of errors and exceptions (try and except)


Today!
Hello
Pandas!

Data Structures

Variables
Lists
Dictionaries
Tuples
Sets

Data Structures

Variables
Lists
Dictionaries
Tuples
Sets

ndarray

Series
DataFrames

Data Structures

Variables
Lists
Dictionaries
Tuples
Sets

ndarray

More about this in 2 weeks

Series
DataFrames

Today!

Introduction to Pandas
Data Structures and I/O Operations

Welcome to Pandas

The first step is to load Pandas, we will use the alias pd

```
import pandas as pd
```

PD Series

The first step is to load Pandas, we will use the alias pd

```
import pandas as pd
```

A **Series** is a one-dimensional labelled array-like object. It is capable of holding any data type, e.g. integers, floats, strings, Python objects, and so on. It can be seen as a data structure with two arrays: one functioning as the index, i.e. the labels, and the other one contains the actual data.

```
obj = pd.Series([11,28,72,3,5,8])
```

```
obj
```

```
0    11
1    28
2    72
3     3
4     5
5     8
dtype: int64
```

PD Series

The first step is to load Pandas, we will use the alias pd

```
import pandas as pd
```

A **Series** is a one-dimensional labelled array-like object. It is capable of holding any data type, e.g. integers, floats, strings, Python objects, and so on. It can be seen as a data structure with two arrays: one functioning as the index, i.e. the labels, and the other one contains the actual data.

```
obj = pd.Series([11,28,72,3,5,8])
```

```
obj
```

Index

0	11
1	28
2	72
3	3
4	5
5	8

Values

```
dtype: int64
```

```
obj.array
```

```
<NumpyExtensionArray>
[11, 28, 72, 3, 5, 8]
Length: 6, dtype: int64
```

```
obj.index
```

```
RangeIndex(start=0, stop=6, step=1)
```

PD Series

We can provide a customised index list

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
```

```
print(S)
```

```
apples      20
oranges     33
cherries    52
pears       10
dtype: int64
```

PD Series

We can provide a customised index list

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
```

```
print(S)
```

```
apples      20
oranges     33
cherries    52
pears       10
dtype: int64
```

A useful Series feature for many applications is that it automatically aligns by index label in arithmetic operations

```
S2 = pd.Series([17, 13, 31, 32], index=fruits)
print(S + S2)
print("sum of S: ", sum(S))
```

```
apples      37
oranges     46
cherries    83
pears       42
dtype: int64
sum of S:  115
```

PD Series

We can provide a customised index list

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
```

```
print(S)
```

```
apples      20
oranges     33
cherries    52
pears       10
dtype: int64
```

A useful Series feature for many applications is that it automatically aligns by index label in arithmetic operations

```
fruits2 = ['raspberries', 'oranges', 'cherries', 'pears']
S2 = pd.Series([17, 13, 31, 32], index=fruits2)
print(S + S2)
```

```
apples          NaN
cherries       83.0
oranges        46.0
pears          42.0
raspberries     NaN
dtype: float64
```

You can think about this
as being similar to a join operation!

PD Series

We can provide a customised index list

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
```

```
print(S)
```

```
apples      20
oranges     33
cherries    52
pears       10
dtype: int64
```

```
fruits2 = ['raspberries', 'oranges', 'cherries', 'pears']
S2 = pd.Series([17, 13, 31, 32], index=fruits2)
print(S + S2)
```

```
apples          NaN
cherries       83.0
oranges        46.0
pears          42.0
raspberries     NaN
dtype: float64
```

Not a Number

PD Series

We can access elements of a Series just like a Dictionary

```
print(S)
```

```
apples      20
oranges     33
cherries    52
pears       10
dtype: int64
```

```
print(S['apples'])
```

```
20
```

```
S['apples'] = 25
print(S)
```

```
apples      25
oranges     33
cherries    52
pears       10
dtype: int64
```

PD Series

Several Operations are broadcasted element-wise

```
print(S)
```

```
apples      25
oranges     33
cherries    52
pears       10
dtype: int64
```

```
print(S+2)
```

```
apples      27
oranges     35
cherries    54
pears       12
dtype: int64
```

```
print(np.sin(S))
```

```
apples     -0.132352
oranges     0.999912
cherries    0.986628
pears      -0.544021
dtype: float64
```

PD Series

We can use the Apply(), to apply a function to each value in Series

```
print(S)
```

```
apples      25
oranges     33
cherries    52
pears       10
dtype: int64
```

```
S.apply(lambda x: x if x > 25 else -1*x )
```

```
apples     -25
oranges     33
cherries    52
pears      -10
dtype: int64
```

Apply(), is a special type of Map() that can operate on the entire Series.
Map applies a function element-wise.

PD Series

We can use the Apply(), to apply a function to each value in Series

```
print(S)
```

```
apples      20
oranges     33
cherries    52
pears       10
dtype: int64
```

```
S.apply(lambda x: x if x > 25 else -1*x )
```

```
apples     -25
oranges     33
cherries    52
pears      -10
dtype: int64
```

What's the difference between apply and map?

PD Series

Filter using booleans

```
print(S)
```

```
apples      25
oranges     33
cherries    52
pears       10
dtype: int64
```

```
print(S[S>25])
```

```
oranges     33
cherries    52
dtype: int64
```

Check if key exists

```
'apples' in S
```

```
True
```

PD Series

We can also initiate a Series using a Dictionary

```
cities = {"London":   8615246,
          "Berlin":   3562166,
          "Madrid":   3165235,
          "Rome":     2874038,
          "Paris":    2273305,
          "Vienna":   1805681,
          "Bucharest":1803425,
          "Hamburg":  1760433,
          "Budapest": 1754000,
          "Warsaw":   1740119,
          "Barcelona":1602386,
          "Munich":   1493900,
          "Milan":    1350680}
city_series = pd.Series(cities)
print(city_series)
```

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
dtype: int64
```

PD Series

Or using another series, while forcing the index list

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]
my_city_series = pd.Series(cities,
                           index=my_cities)
my_city_series
```

```
London       8615246.0
Paris        2273305.0
Zurich             NaN
Berlin       3562166.0
Stuttgart          NaN
Hamburg      1760433.0
dtype: float64
```

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
dtype: int64
```

PD Series

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
dtype: int64
```

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]
my_city_series = pd.Series(cities,
                           index=my_cities)
my_city_series
```

```
London       8615246.0
Paris        2273305.0
Zurich             NaN
Berlin       3562166.0
Stuttgart          NaN
Hamburg      1760433.0
dtype: float64
```

```
print(my_city_series.isnull())
```

```
London       False
Paris        False
Zurich        True
Berlin       False
Stuttgart     True
Hamburg      False
dtype: bool
```

PD Series

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
dtype: int64
```

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]
my_city_series = pd.Series(cities,
                           index=my_cities)
my_city_series
```

```
London       8615246.0
Paris        2273305.0
Zurich             NaN
Berlin       3562166.0
Stuttgart          NaN
Hamburg      1760433.0
dtype: float64
```

```
print(my_city_series.notnull())
```

```
London        True
Paris         True
Zurich       False
Berlin        True
Stuttgart    False
Hamburg       True
dtype: bool
```

PD Series

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
dtype: int64
```

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]
my_city_series = pd.Series(cities,
                           index=my_cities)
my_city_series
```

```
London       8615246.0
Paris        2273305.0
Zurich             NaN
Berlin       3562166.0
Stuttgart          NaN
Hamburg      1760433.0
dtype: float64
```

```
print(my_city_series.fillna(0))
```

```
London       8615246.0
Paris        2273305.0
Zurich             0.0
Berlin       3562166.0
Stuttgart          0.0
Hamburg      1760433.0
dtype: float64
```

PD Series

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
dtype: int64
```

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]
my_city_series = pd.Series(cities,
                           index=my_cities)
my_city_series
```

```
London       8615246.0
Paris        2273305.0
Zurich             NaN
Berlin       3562166.0
Stuttgart          NaN
Hamburg      1760433.0
dtype: float64
```

```
print(my_city_series.fillna(0))
```

```
London       8615246.0
Paris        2273305.0
Zurich             0.0
Berlin       3562166.0
Stuttgart          0.0
Hamburg      1760433.0
dtype: float64
```

PD Series

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
dtype: int64
```

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]
my_city_series = pd.Series(cities,
                           index=my_cities)
my_city_series
```

```
London       8615246.0
Paris        2273305.0
Zurich             NaN
Berlin       3562166.0
Stuttgart          NaN
Hamburg      1760433.0
dtype: float64
```

```
print(my_city_series.fillna(0).astype(int))
```

```
London       8615246
Paris        2273305
Zurich             0
Berlin       3562166
Stuttgart          0
Hamburg      1760433
dtype: int64
```

PD Series

Both the Series object itself and its index have a name attribute, which integrates with other areas of pandas functionality

```
obj = pd.Series([4, 7, -5, 3],index=['foo','faa','fee','bar'])
print(obj)
```

```
foo    4
faa    7
fee   -5
bar    3
dtype: int64
```

```
obj.name = 'foobar'
obj.index.name = 'things'
```

```
print(obj)
```

```
things
foo    4
faa    7
fee   -5
bar    3
Name: foobar, dtype: int64
```

PD Index Objects

are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or sequence of labels used when constructing, for instance, a Series is internally coverted to an Index.

```
obj = pd.Series([4, 7, -5, 3],index=['foo','faa','fee','bar'])
print(obj)
```

```
foo    4
faa    7
fee   -5
bar    3
dtype: int64
```

```
obj.index
```

```
Index(['foo', 'faa', 'fee', 'bar'], dtype='object')
```

PD Index Objects

are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or sequence of labels used when constructing, for instance, a Series is internally coverted to an Index.

```
obj = pd.Series([4, 7, -5, 3],index=['foo','faa','fee','bar'])
print(obj)
```

```
foo    4
faa    7
fee   -5
bar    3
dtype: int64
```

```
obj.index
```

```
Index(['foo', 'faa', 'fee', 'bar'], dtype='object')
```

Index Objects Properties

- Immutable

Thus they cannot be modified by users

```
obj.index[1] = 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[15], line 1
----> 1 obj.index[1] = 2

File ~/.pyenv/versions/3.13.5/lib/python3.13/site-packages/pandas/core/indexes/base.py:5383, in Index.__setitem__(self, ke
   5381 @final
   5382 def __setitem__(self, key, value) -> None:
-> 5383     raise TypeError("Index does not support mutable operations")

TypeError: Index does not support mutable operations
```

PD Index Objects

are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or sequence of labels used when constructing, for instance, a Series is internally coverted to an Index.

```
obj = pd.Series([4, 7, -5, 3],index=['foo','faa','fee','bar'])
print(obj)
```

```
foo    4
faa    7
fee   -5
bar    3
dtype: int64
```

```
obj.index
```

```
Index(['foo', 'faa', 'fee', 'bar'], dtype='object')
```

Index Objects Properties

- Immutable

Thus they cannot be modified by users

but makes it safer to share Index objects among data structures

PD Index Objects

are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or sequence of labels used when constructing, for instance, a Series is internally coverted to an Index.

```
obj = pd.Series([4, 7, -5, 3],index=['foo','faa','fee','bar'])
print(obj)
```

```
foo    4
faa    7
fee   -5
bar    3
dtype: int64
```

```
obj.index
```

```
Index(['foo', 'faa', 'fee', 'bar'], dtype='object')
```

Index Objects Properties

- Immutable
- Behaves like a Fixed-size set

```
'foo' in obj.index
```

```
True
```

```
'foobar' in obj.index
```

```
False
```

PD Index Objects

are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or sequence of labels used when constructing, for instance, a Series is internally coverted to an Index.

```
obj = pd.Series([4, 7, -5, 3],index=['foo','faa','fee','bar'])
print(obj)
```

```
foo    4
faa    7
fee   -5
bar    3
dtype: int64
```

```
obj.index
```

```
Index(['foo', 'faa', 'fee', 'bar'], dtype='object')
```

Index Objects Properties

- Immutable
- Behaves like a Fixed-size set
- Can contain duplicated labels

```
pd.Index(["foo", "foo", "bar", "bar"])
```

```
Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Selections with duplicate labels will select all occurrences of that label

PD Index Objects

are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or sequence of labels used when constructing, for instance, a Series is internally coverted to an Index.

Method/Property	Description
`append()`	Concatenate with additional Index objects, producing a new Index
`difference()`	Compute set difference as an Index
`intersection()`	Compute set intersection
`union()`	Compute set union
`isin()`	Compute Boolean array indicating whether each value is contained in the passed collection
`delete()`	Compute new Index with element at Index `i` deleted
`drop()`	Compute new Index by deleting passed values
`insert()`	Compute new Index by inserting element at Index `i`
`is_monotonic`	Returns `True` if each element is greater than or equal to the previous element
`is_unique`	Returns `True` if the Index has no duplicate values
`unique()`	Compute the array of unique values in the Index

Several useful methods associated with pd Index Objects

PD Objects (Series)

name	string
Index Integer Position	like python lists and tuples
Index Labels	Array-like, sequence of labels that allow us to fetch elements like in dictionaries (can have a name)
Array of Values	Data values

PD DataFrames

The underlying idea of a **DataFrame** is based on spreadsheets. We can see the data structure of a DataFrame as tabular and spreadsheet-like. A DataFrame logically corresponds to a "sheet" of an Excel document. A DataFrame has both a row and a column index.

PD DataFrames

Index

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	...
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	...
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	...
2	37	services	married	high.school	no	yes	no	telephone	may	mon	...
3	40	admin.	married	basic.6y	no	no	no	telephone	may	mon	...
4	56	services	married	high.school	no	no	yes	telephone	may	mon	...

Columns

Data

5 rows × 21 columns

```
df.dtypes
```

```
age                int64
job               object
marital           object
education         object
default           object
housing           object
loan              object
```

dtype

PD DataFrames

The underlying idea of a **DataFrame** is based on spreadsheets. We can see the data structure of a DataFrame as tabular and spreadsheet-like. A DataFrame logically corresponds to a "sheet" of an Excel document. A DataFrame has both a row and a column index.

```
years = range(2014, 2018)
shop1 = pd.Series([2409.14, 2941.01, 3496.83, 3119.55], index=years)
shop2 = pd.Series([1203.45, 3441.62, 3007.83, 3619.53], index=years)
shop3 = pd.Series([3412.12, 3491.16, 3457.19, 1963.10], index=years)
```

```
pd.concat([shop1, shop2, shop3])
```

```
2014    2409.14
2015    2941.01
2016    3496.83
2017    3119.55
2014    1203.45
2015    3441.62
2016    3007.83
2017    3619.53
2014    3412.12
2015    3491.16
2016    3457.19
2017    1963.10
dtype: float64
```

PD DataFrames

```
shops_df = pd.concat([shop1, shop2, shop3], axis=1)
shops_df
```

	0	1	2
2014	2409.14	1203.45	3412.12
2015	2941.01	3441.62	3491.16
2016	3496.83	3007.83	3457.19
2017	3119.55	3619.53	1963.10

PD DataFrames

```
shops_df = pd.concat([shop1, shop2, shop3], axis=1)
shops_df
```

	0	1	2
2014	2409.14	1203.45	3412.12
2015	2941.01	3441.62	3491.16
2016	3496.83	3007.83	3457.19
2017	3119.55	3619.53	1963.10

The axis to concatenate along, default is 0

PD DataFrames

```
shops_df = pd.concat([shop1, shop2, shop3], axis=1)
shops_df
```

```
cities = ["Zürich", "Winterthur", "Freiburg"]
shops_df.columns = cities
print(shops_df)
# alternative way: give names to series:
shop1.name = "Zürich"
shop2.name = "Winterthur"
shop3.name = "Freiburg"
print("------")
shops_df2 = pd.concat([shop1, shop2, shop3], axis=1)
print(shops_df2)
```

```
      Zürich  Winterthur  Freiburg
2014  2409.14     1203.45   3412.12
2015  2941.01     3441.62   3491.16
2016  3496.83     3007.83   3457.19
2017  3119.55     3619.53   1963.10
------
      Zürich  Winterthur  Freiburg
2014  2409.14     1203.45   3412.12
2015  2941.01     3441.62   3491.16
2016  3496.83     3007.83   3457.19
2017  3119.55     3619.53   1963.10
```

PD DataFrames

```
cities = ["Zürich", "Winterthur", "Freiburg"]
shops_df.columns = cities
print(shops_df)
# alternative way: give names to series:
shop1.name = "Zürich"
shop2.name = "Winterthur"
shop3.name = "Freiburg"
print("------")
shops_df2 = pd.concat([shop1, shop2, shop3], axis=1)
print(shops_df2)
```

```
       Zürich  Winterthur  Freiburg
2014  2409.14     1203.45   3412.12
2015  2941.01     3441.62   3491.16
2016  3496.83     3007.83   3457.19
2017  3119.55     3619.53   1963.10
------
       Zürich  Winterthur  Freiburg
2014  2409.14     1203.45   3412.12
2015  2941.01     3441.62   3491.16
2016  3496.83     3007.83   3457.19
2017  3119.55     3619.53   1963.10
```

```
print(type(shops_df))
```

```
<class 'pandas.core.frame.DataFrame'>
```

PD DataFrames

Like with Series, we can create a DataFrame from a Dictionary

```
cities = {"name": ["London", "Berlin", "Madrid", "Rome",
                   "Paris", "Vienna", "Bucharest", "Hamburg",
                   "Budapest", "Warsaw", "Barcelona",
                   "Munich", "Milan"],
          "population": [8615246, 3562166, 3165235, 2874038,
                         2273305, 1805681, 1803425, 1760433,
                         1754000, 1740119, 1602386, 1493900,
                         1350680],
          "country": ["England", "Germany", "Spain", "Italy",
                      "France", "Austria", "Romania",
                      "Germany", "Hungary", "Poland", "Spain",
                      "Germany", "Italy"]}
city_frame = pd.DataFrame(cities)
city_frame
```

	name	population	country
0	London	8615246	England
1	Berlin	3562166	Germany
2	Madrid	3165235	Spain
3	Rome	2874038	Italy
4	Paris	2273305	France
5	Vienna	1805681	Austria
6	Bucharest	1803425	Romania
7	Hamburg	1760433	Germany
8	Budapest	1754000	Hungary
9	Warsaw	1740119	Poland
10	Barcelona	1602386	Spain
11	Munich	1493900	Germany
12	Milan	1350680	Italy

PD DataFrames

We can retrieve the columns names

```
city_frame.columns.values
```

```
array(['name', 'population', 'country'], dtype=object)
```

PD DataFrames

We can retrieve the columns names

```
city_frame.columns.values
```

```
array(['name', 'population', 'country'], dtype=object)
```

We can define names for the row Index

```
ordinals = ["first", "second", "third", "fourth",
            "fifth", "sixth", "seventh", "eigth",
            "ninth", "tenth", "eleventh", "twelvth",
            "thirteenth"]
city_frame = pd.DataFrame(cities, index=ordinals)
city_frame
```

	name	population	country
first	London	8615246	England
second	Berlin	3562166	Germany
third	Madrid	3165235	Spain
fourth	Rome	2874038	Italy
fifth	Paris	2273305	France
sixth	Vienna	1805681	Austria
seventh	Bucharest	1803425	Romania
eigth	Hamburg	1760433	Germany
ninth	Budapest	1754000	Hungary
tenth	Warsaw	1740119	Poland

PD DataFrames

We can rearrange the order of the columns

```
city_frame = pd.DataFrame(cities,
                          columns=["name", "country", "population"])
city_frame
```

	name	country	population
0	London	England	8615246
1	Berlin	Germany	3562166
2	Madrid	Spain	3165235
3	Rome	Italy	2874038
4	Paris	France	2273305
5	Vienna	Austria	1805681
6	Bucharest	Romania	1803425
7	Hamburg	Germany	1760433
8	Budapest	Hungary	1754000
9	Warsaw	Poland	1740119
10	Barcelona	Spain	1602386
11	Munich	Germany	1493900
12	Milan	Italy	1350680

PD DataFrames

We can rearrange the order of the columns.

```
city_frame.reindex(["country", "name", "population"])
city_frame
```

This time using reindex

	name	country	population
0	London	England	8615246
1	Berlin	Germany	3562166
2	Madrid	Spain	3165235
3	Rome	Italy	2874038
4	Paris	France	2273305
5	Vienna	Austria	1805681
6	Bucharest	Romania	1803425
7	Hamburg	Germany	1760433
8	Budapest	Hungary	1754000
9	Warsaw	Poland	1740119
10	Barcelona	Spain	1602386
11	Munich	Germany	1493900
12	Milan	Italy	1350680

PD DataFrames

We can also rename the columns

```
city_frame.rename(columns={
    "name":"nome",
    "country":"pais",
    "population":"populacao"
}, inplace=True)
city_frame
```

Whether to return a new DataFrame.
If True then value of copy is ignored

	nome	pais	populacao
0	London	England	8615246
1	Berlin	Germany	3562166
2	Madrid	Spain	3165235
3	Rome	Italy	2874038
4	Paris	France	2273305
5	Vienna	Austria	1805681
6	Bucharest	Romania	1803425
7	Hamburg	Germany	1760433
8	Budapest	Hungary	1754000
9	Warsaw	Poland	1740119
10	Barcelona	Spain	1602386
11	Munich	Germany	1493900
12	Milan	Italy	1350680

PD DataFrames

Sometimes a more useful index would use the country name as the index, i.e. the list value associated to the key "country" of our cities dictionary:

```
city_frame = pd.DataFrame(cities,
                          columns=["name", "population"],
                          index=cities["country"])
city_frame
```

	name	population
England	London	8615246
Germany	Berlin	3562166
Spain	Madrid	3165235
Italy	Rome	2874038
France	Paris	2273305
Austria	Vienna	1805681
Romania	Bucharest	1803425
Germany	Hamburg	1760433
Hungary	Budapest	1754000
Poland	Warsaw	1740119
Spain	Barcelona	1602386
Germany	Munich	1493900
Italy	Milan	1350680

Alternatively we can do

```
city_frame2 = city_frame.set_index("country")
```

```
city_frame.set_index("country", inplace=True)
```

PD DataFrames

Accessing and Indexing of Rows

```
city_frame = pd.DataFrame(cities,
                          columns=("name", "population"),
                          index=cities["country"])
print(city_frame.loc["Germany"])
```

```
            name  population
Germany   Berlin     3562166
Germany  Hamburg     1760433
Germany   Munich     1493900
```

```
print(city_frame.loc[["Germany", "France"]])
```

```
            name  population
Germany   Berlin     3562166
Germany  Hamburg     1760433
Germany   Munich     1493900
France     Paris     2273305
```

```
print(city_frame.loc[city_frame.population>2000000])
```

```
            name  population
England  London     8615246
Germany  Berlin     3562166
Spain    Madrid     3165235
Italy      Rome     2874038
France    Paris     2273305
```

PD DataFrames

Summing over all the columns

```
print(city_frame.sum())
```

```
name          LondonBerlinMadridRomeParisViennaBucharestHamb...
population                                             33800614
dtype: object
```

Summing over all values in a single column

```
city_frame["population"].sum()
```

```
33800614
```

Cumulative Sum

```
x = city_frame["population"].cumsum()
print(x)
```

```
England     8615246
Germany    12177412
Spain      15342647
Italy      18216685
France     20489990
Austria    22295671
Romania    24099096
Germany    25859529
Hungary    27613529
Poland     29353648
Spain      30956034
Germany    32449934
Italy      33800614
Name: population, dtype: int64
```

PD DataFrames

Let's take x as a Pandas Series. We can reassign the previously calculated cumulative sums to the population column:

```
city_frame["population"] = x
print(city_frame)
```

```
                name  population
England       London     8615246
Germany       Berlin    12177412
Spain         Madrid    15342647
Italy           Rome    18216685
France         Paris    20489990
Austria       Vienna    22295671
Romania    Bucharest    24099096
Germany      Hamburg    25859529
Hungary     Budapest    27613529
Poland       Warsaw    29353648
Spain     Barcelona    30956034
Germany      Munich    32449934
Italy         Milan    33800614
```

```
x = city_frame["population"].cumsum()
```

PD DataFrames

Instead of replacing the values of the population column with the cumulative sum, we want to add the cumulative population sum as a new column with the name "cum_population".

```
city_frame = pd.DataFrame(cities,
                          columns=["country",
                                   "population",
                                   "cum_population"],
                          index=cities["name"])
city_frame
```

	country	population	cum_population
London	England	8615246	NaN
Berlin	Germany	3562166	NaN
Madrid	Spain	3165235	NaN
Rome	Italy	2874038	NaN
Paris	France	2273305	NaN
Vienna	Austria	1805681	NaN
Bucharest	Romania	1803425	NaN
Hamburg	Germany	1760433	NaN
Budapest	Hungary	1754000	NaN
Warsaw	Poland	1740119	NaN
Barcelona	Spain	1602386	NaN
Munich	Germany	1493900	NaN
Milan	Italy	1350680	NaN

PD DataFrames

We can assign now the cumulative sums to this column:

```
city_frame["cum_population"] = city_frame["population"].cumsum()
city_frame
```

	country	population	cum_population
London	England	8615246	8615246
Berlin	Germany	3562166	12177412
Madrid	Spain	3165235	15342647
Rome	Italy	2874038	18216685
Paris	France	2273305	20489990
Vienna	Austria	1805681	22295671
Bucharest	Romania	1803425	24099096
Hamburg	Germany	1760433	25859529
Budapest	Hungary	1754000	27613529
Warsaw	Poland	1740119	29353648
Barcelona	Spain	1602386	30956034
Munich	Germany	1493900	32449934
Milan	Italy	1350680	33800614

PD DataFrames

We can also include a column name which was not contained in the dictionary used to create the DataFrame. In this case, all the values of this column will be set to NaN:

```
city_frame = pd.DataFrame(cities,
                          columns=["country",
                                   "area",
                                   "population"],
                          index=cities["name"])
print(city_frame)
```

```
           country area  population
London     England  NaN     8615246
Berlin     Germany  NaN     3562166
Madrid       Spain  NaN     3165235
Rome         Italy  NaN     2874038
Paris       France  NaN     2273305
Vienna     Austria  NaN     1805681
Bucharest  Romania  NaN     1803425
Hamburg    Germany  NaN     1760433
Budapest   Hungary  NaN     1754000
Warsaw      Poland  NaN     1740119
Barcelona    Spain  NaN     1602386
Munich     Germany  NaN     1493900
Milan        Italy  NaN     1350680
```

PD DataFrames

Accessing Columns can be done by passing a key inside [] like in a Dictionary

```
print(city_frame["population"])
```

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
Name: population, dtype: int64
```

or by calling the column as an attribute of the DataFrame

```
print(city_frame.population)
```

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
Name: population, dtype: int64
```

PD DataFrames

Assigning new values to Columns by calling the column name it fills all rows with the same value

```
city_frame["area"] = 1572
print(city_frame.head())
```

```
        country  area  population
London  England  1572     8615246
Berlin  Germany  1572     3562166
Madrid    Spain  1572     3165235
Rome      Italy  1572     2874038
Paris    France  1572     2273305
```

by assigning a list of same size, it fills up the entire column with the values we pass

```
# area in square km:
area = [1572, 891.85, 605.77, 1285,
        105.4, 414.6, 228, 755,
        525.2, 517, 101.9, 310.4,
        181.8]
# area could have been designed as a list, a Series, an array or a scalar
city_frame["area"] = area
print(city_frame.head())
```

```
        country     area  population
London  England  1572.00     8615246
Berlin  Germany   891.85     3562166
Madrid    Spain   605.77     3165235
Rome      Italy  1285.00     2874038
Paris    France   105.40     2273305
```

PD DataFrames

We can create a DataFrame from nested dictionaries

```
growth = {"Switzerland": {"2010": 3.0, "2011": 1.8, "2012": 1.1, "2013": 1.9},
          "Germany": {"2010": 4.1, "2011": 3.6, "2012":0.4, "2013": 0.1},
          "France": {"2010":2.0,  "2011":2.1, "2012": 0.3, "2013": 0.3},
          "Greece": {"2010":-5.4, "2011":-8.9, "2012":-6.6, "2013":-3.3},
          "Italy": {"2010":1.7, "2011":0.6, "2012":-2.3, "2013":-1.9}
          }
growth_frame = pd.DataFrame(growth)
growth_frame
```

	Switzerland	Germany	France	Greece	Italy
2010	3.0	4.1	2.0	-5.4	1.7
2011	1.8	3.6	2.1	-8.9	0.6
2012	1.1	0.4	0.3	-6.6	-2.3
2013	1.9	0.1	0.3	-3.3	-1.9

PD DataFrames

	Switzerland	Germany	France	Greece	Italy
2010	3.0	4.1	2.0	-5.4	1.7
2011	1.8	3.6	2.1	-8.9	0.6
2012	1.1	0.4	0.3	-6.6	-2.3
2013	1.9	0.1	0.3	-3.3	-1.9

the attribute .T returns the transpose of a table

```
growth_frame.T
```

	2010	2011	2012	2013
Switzerland	3.0	1.8	1.1	1.9
Germany	4.1	3.6	0.4	0.1
France	2.0	2.1	0.3	0.3
Greece	-5.4	-8.9	-6.6	-3.3
Italy	1.7	0.6	-2.3	-1.9

PD Loading data

What about loading data from files?
It couldn't be easier with Pandas!
it takes one line of code!

```
import pandas as pd
exchange_rates = pd.read_csv("dollar_euro.txt",
                             sep="\t")
print(exchange_rates)
```

```
    Year   Average  Min USD/EUR  Max USD/EUR  Working days
0   2016  0.901696     0.864379     0.959785           247
1   2015  0.901896     0.830358     0.947688           256
2   2014  0.753941     0.716692     0.823655           255
3   2013  0.753234     0.723903     0.783208           255
4   2012  0.778848     0.743273     0.827198           256
5   2011  0.719219     0.671953     0.775855           257
6   2010  0.755883     0.686672     0.837381           258
7   2009  0.718968     0.661376     0.796495           256
8   2008  0.683499     0.625391     0.802568           256
9   2007  0.730754     0.672314     0.775615           255
10  2006  0.797153     0.750131     0.845594           255
11  2005  0.805097     0.740357     0.857118           257
12  2004  0.804828     0.733514     0.847314           259
13  2003  0.885766     0.791766     0.963670           255
14  2002  1.060945     0.953562     1.165773           255
15  2001  1.117587     1.047669     1.192748           255
16  2000  1.085899     0.962649     1.211827           255
17  1999  0.939475     0.848176     0.998502           261
```

PD Loading data

```
file_path = ''
file_name = 'bank-additional-full.csv'
df = pd.read_csv(
    file_path+file_name,
    delimiter = ';',
    header = 0,
    decimal = '.',
    quotechar = '"'
)
```

```
type(df)
```

```
pandas.core.frame.DataFrame
```

```
df.shape
```

```
(41188, 21)
```

Normal CSV

Delimiter: ‘,’
EOL: New line (‘\n’)
Decimal: ‘.’
Quote Char: “”

Portuguese CSV

Delimiter: ‘;’
EOL: New line (‘\n’)
Decimal: ‘,’
Quote Char: “”

Note: in Portugal the default decimal separator is ‘,’, so it cannot be used to delimit columns.

Normal TSV

Delimiter: Tab (‘\t’)
EOL: New line (‘\n’)
Decimal: ‘,’
Quote Char: “”

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

PD Loading data

```
column_names = ["Country"] + list(range(2002, 2013))
male_pop = pd.read_csv("countries_male_population.csv",
                       header=None,
                       index_col=0,
                       names=column_names)
female_pop = pd.read_csv("countries_female_population.csv",
                         header=None,
                         index_col=0,
                         names=column_names)
population = male_pop + female_pop
```

```
population.head()
```

Country	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012
Australia	19506266.0	19746894	19981026	20242876	20514836	20889244	21321834	21776770	22248508	22521494	22561608
Austria	7919134.0	7818240	7899650	7972592	8038708	8074342	8108428	8136094	8158186	8190674	8236070
Belgium	10084576.0	10133770	10174352	10222650	10287642	10362816	10448618	10537302	10624442	10740468	10827602
Canada	NaN	31064876	31077144	31685574	31991164	32341446	32652282	33049008	33053352	33652244	34227082
Czech Republic	10011016.0	9933412	9949480	9961826	10005296	10052368	10165868	10272754	10314394	10337598	10316420

```
population.to_csv("data1/countries_total_population.csv")
```

What about saving data to files?
Again, It couldn't be made easier with Pandas! it takes one line of code!

Indexing and Selection

Accessing elements in Series and Dataframes.

>> It is confusing for people experimented with other programming languages. <<

.loc[] is primarily label based, but may also be used with a boolean array. Will raise KeyError when the items are not found

.iloc[] is primarily integer position based (from 0 to length-1of the axis), but may also be used with a boolean array. Will raise IndexError if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing.

[] is also possible, its primary function is to select out lower-dimensional slices. The following tables indicates the return type when indexing pandas objects with []:

Object Type	Selection	Return Value Type
Series	`series[label]`	scalar value
DataFrame	`frame[colname]`	`Series` corresponding to colname

Indexing and Selection on Series

let's take two examples

```
obj1 = pd.Series([1,2,3,4], index=[2, 0, 1, 3])
obj2 = pd.Series([1,2,3,4], index=["a", "b", "c", "d"])
```

```
obj1
```

```
2    1
0    2
1    3
3    4
dtype: int64
```

```
obj2
```

```
a    1
b    2
c    3
d    4
dtype: int64
```

as expected we can do stuff like

```
obj2["b"]
```

```
np.int64(2)
```

fetch values like a dictionary

```
obj2[2:4]
```

```
c    3
d    4
dtype: int64
```

slice by index position

```
obj2[["b", "a", "d"]]
```

```
b    2
a    1
d    4
dtype: int64
```

fetch multiple entries

```
obj2[obj2 < 2]
```

```
a    1
dtype: int64
```

select based on condition

Indexing and Selection on Series

let's take two examples

```
obj1 = pd.Series([1,2,3,4], index=[2, 0, 1, 3])
obj2 = pd.Series([1,2,3,4], index=["a", "b", "c", "d"])
```

```
obj1
```

```
2    1
0    2
1    3
3    4
dtype: int64
```

```
obj2
```

```
a    1
b    2
c    3
d    4
dtype: int64
```

as expected we can do stuff like

```
obj2["b"]
```

fetch values like a dictionary

```
np.int64(2)
```

```
obj2[2:4]
```

slice by index position

```
c    3
d    4
dtype:
```

```
obj2["b":"d"]
```

```
b    2
c    3
d    4
dtype: int64
```

slice by labels!?

```
obj2[["
```

multiple entries

```
b    2
a    1
d    4
dtype: int64
```

```
obj2[obj2 < 2]
```

select based on condition

```
a    1
dtype: int64
```

Indexing and Selection on Series

let's take two examples

```
obj1 = pd.Series([1,2,3,4], index=[2, 0, 1, 3])
obj2 = pd.Series([1,2,3,4], index=["a", "b", "c", "d"])
```

```
obj1
```

```
2    1
0    2
1    3
3    4
dtype: int64
```

```
obj2
```

```
a    1
b    2
c    3
d    4
dtype: int64
```

however the recommended behavior is to use **.loc[]**

```
obj2.loc[["b", "a", "d"]]
```

```
b    2
a    1
d    4
dtype: int64
```

to fech elements by label

differentiate the access by index

Indexing and Selection on Series

let's take two examples

```
obj1 = pd.Series([1,2,3,4], index=[2, 0, 1, 3])
obj2 = pd.Series([1,2,3,4], index=["a", "b", "c", "d"])
```

```
obj1
```

```
2    1
0    2
1    3
3    4
dtype: int64
```

```
obj2
```

```
a    1
b    2
c    3
d    4
dtype: int64
```

however the recommended behavior is to use **.loc[]**

```
obj2.loc[["b", "a", "d"]]
```

```
b    2
a    1
d    4
dtype: int64
```

to fech elements by label
differentiate the access by index

<u>big confusion here</u>

```
obj1[[0,1,2]]
```

```
0    2
1    3
2    1
dtype: int64
```

here we are fetching **by labels**

```
obj2[[0,1,2]]
```

```
a    1
b    2
c    3
dtype: int64
```

here **by index position**

Indexing and Selection on Series

let's take two examples

```
obj1 = pd.Series([1,2,3,4], index=[2, 0, 1, 3])
obj2 = pd.Series([1,2,3,4], index=["a", "b", "c", "d"])
```

```
obj1
```

```
2    1
0    2
1    3
3    4
dtype: int64
```

```
obj2
```

```
a    1
b    2
c    3
d    4
dtype: int64
```

however the recommended behavior is to use **.loc[]**

```
obj2.loc[["b", "a", "d"]]
```

```
b    2
a    1
d    4
dtype: int64
```

to fech elements by label
differentiate the access by index

.loc[] only fetches data by label.

.iloc[] fetches by index position

and these are the recommended positional operators!

Indexing and Selection on Dataframes

let's take one example

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=["Ohio", "Colorado", "Utah", "New York"],
                    columns=["one", "two", "three", "four"]
                   )
```

```
data
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

positional operator []

```
data['two']
```

```
Ohio          1
Colorado      5
Utah          9
New York     13
Name: two, dtype: int64
```

single label fetches
one columns as a Series!

```
data[['two','three']]
```

	two	three
Ohio	1	2
Colorado	5	6
Utah	9	10
New York	13	14

list of labels fetches
multiple columns as a dataframe!

Indexing and Selection on Dataframes

let's take one example

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=["Ohio", "Colorado", "Utah", "New York"],
                    columns=["one", "two", "three", "four"]
                   )
```

```
data
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

positional operator .loc[]

```
data.loc['Colorado']
```

```
one      4
two      5
three    6
four     7
Name: Colorado, dtype: int64
```

fetches rows that match the label!

```
data.loc['Colorado',['two','three']]
```

```
two      5
three    6
Name: Colorado, dtype: int64
```

two arguments separated by ',' to select by row label and column label

```
data.loc[:'Colorado',['two','three']]
```

	two	three
Ohio	1	2
Colorado	5	6

slices apply!

Indexing and Selection on Dataframes

let's take one example

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=["Ohio", "Colorado", "Utah", "New York"],
                    columns=["one", "two", "three", "four"]
                   )
```

```
data
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

positional operator .iloc[]

```
data.iloc[[2, 1]]
```

	one	two	three	four
Utah	8	9	10	11
Colorado	4	5	6	7

fetches rows that match the index position!

```
data.iloc[[1, 2], [3, 0, 1]]
```

	four	one	two
Colorado	7	4	5
Utah	11	8	9

two arguments to fetch rows and seelct columns

```
data.iloc[:, :3]
```

	one	two	three
Ohio	0	1	2
Colorado	4	5	6
Utah	8	9	10
New York	12	13	14

slices apply!

Summary of Indexing and Selection

Type	Notes
`df[column]`	Select single column or sequence of columns from the DataFrame; special case conveniences: Boolean array (filter rows), slice (slice rows), or Boolean DataFrame (set values based on some criterion)
`df.loc[rows]`	Select single row or subset of rows from the DataFrame by label
`df.loc[:, cols]`	Select single column or subset of columns by label
`df.loc[rows, cols]`	Select both row(s) and column(s) by label
`df.iloc[rows]`	Select single row or subset of rows from the DataFrame by integer position
`df.iloc[:, cols]`	Select single column or subset of columns by integer position
`df.iloc[rows, cols]`	Select both row(s) and column(s) by integer position
`df.at[row, col]`	Select a single scalar value by row and column label
`df.iat[row, col]`	Select a single scalar value by row and column position (integers)
`reindex` method	Select either rows or columns by labels

PD Final Notes

apply() method can be applied both to series and dataframes where function can be applied both series and individual elements based on the type of function provided.
map() method only works on a pandas series where type of operation to be applied depends on argument passed as a function, dictionary or a list.

applymap() method only works on a pandas dataframe where function is applied on every element individually.

> ***Deprecated since version 2.1.0:*** DataFrame.applymap has been deprecated. Use DataFrame.map instead.

PD Final Notes

Get used to the Documentation of Pandas!

pandas.DataFrame.map

`DataFrame.map(func, na_action=None, **kwargs)` [source]

Apply a function to a Dataframe elementwise.

> *Added in version 2.1.0:* DataFrame.applymap was deprecated and renamed to DataFrame.map.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

Parameters:

func : *callable*

Python function, returns a single value from a single value.

na_action : *{None, 'ignore'}, default None*

If 'ignore', propagate NaN values, without passing them to func.

****kwargs**

Additional keyword arguments to pass as keywords arguments to *func*.

Returns:

DataFrame

Transformed DataFrame.

pandas.DataFrame.apply

`DataFrame.apply(func, axis=0, raw=False, result_type=None, args=(), by_row='compat', engine='python', engine_kwargs=None, **kwargs)` [source]

Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`). By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the *result_type* argument.

Parameters:

func : *function*

Function to apply to each column or row.

axis : *{0 or 'index', 1 or 'columns'}, default 0*

Axis along which the function is applied:

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

raw : *bool, default False*

Determines if row or column is passed as a Series or ndarray object:

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

result_type : *{'expand', 'reduce', 'broadcast', None}, default None*

PD Final Notes

Get used to the Documentation of Pandas!

pandas.DataFrame.map

`DataFrame.map(func, na_action=None, **kwargs)` [source]

Apply a function to a Dataframe elementwise.

pandas.DataFrame.apply

`DataFrame.apply(func, axis=0, raw=False, result_type=None, args=(), by_row='compat', engine='python', engine_kwargs=None, **kwargs)` [source]

Apply a function along an axis of the DataFrame.

pandas.Series.map

`Series.map(arg, na_action=None)` [source]

Map values of Series according to an input mapping or function.

Used for substituting each value in a Series with another value, that may be derived from a function, a `dict` or a `Series`.

Parameters:

arg : *function, collections.abc.Mapping subclass or Series*

Mapping correspondence.

na_action : *{None, 'ignore'}, default None*

If 'ignore', propagate NaN values, without passing them to the mapping correspondence.

Returns:

Series

Same index as caller.

pandas.Series.apply

`Series.apply(func, convert_dtype=<no_default>, args=(), *, by_row='compat', **kwargs)` [source]

Invoke function on values of Series.

Can be ufunc (a NumPy function that applies to the entire Series) or a Python function that only works on single values.

Parameters:

func : *function*

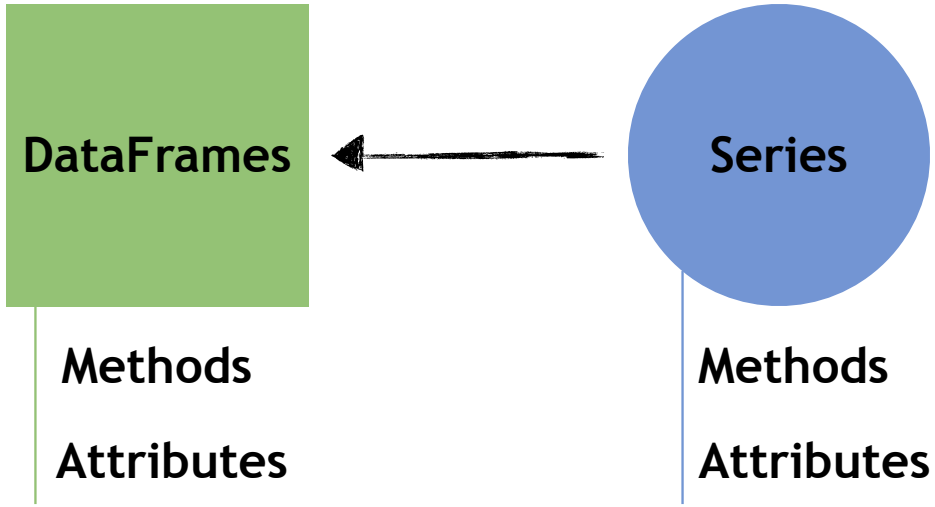
Python function or NumPy ufunc to apply.

convert_dtype : *bool, default True*

Try to find better dtype for elementwise function results. If False, leave as dtype=objec
Note that the dtype is always preserved for some extension array dtypes, such as Categorical.

PD Final Notes

Understand the Objects introduced by Pandas




Next Week
More Pandas!