Programming in Python[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

Academic year 2023/24, I semester

[1]

Lecture XXI: Tabular data

Tabular data

Data are often given/collected as tables: matrices with rows for individual records and columns for the fields of the records. This is especially common in statistics, R has a built-in type for this: the dataframe.

pandas

pandas (Python for data analysis) brings the DataFrame type to Python. It is based on numpy.

- Series: a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index.
- DataFrame: a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet, or a dict of Series objects.

Series

```
import pandas as pd
s = pd.Series(np.random.randn(5), index=["a", "b", "c",
↪  "d", "e"])
```

s is a numpy array of floats, each one has a label.

```
d = {"b": 1, "a": 0, "c": 2}

s = pd.Series(d)
```

The ordering depends on Python and pandas version... The current ones takes the insertion order, but you can provide explicitly the index.

```
d = {"b": 1, "a": 0, "c": 2}

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

Series

A `Series` is convenient because it is a `ndarray` (and can be vectorized) but also a `dict`.

Dataframes

```
d = { "one": pd.Series([1.0, 2.0, 3.0], index=["a",
↪  "b", "c"]),
      "two": pd.Series([1.0, 2.0, 3.0, 4.0],
        ↪  index=["a", "b", "c", "d"]),
    }
```

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

A `DataFrame` has an `index` and a `columns` attribute.
There are many ways of creating `DataFrames`, see the docs.

From csv or spreadsheets

A famous example: Fisher's Iris flowers dataset.
150 records, `"sepal length","sepal width","petal length","petal width","class"`

```
iris = pd.read_csv('iris.csv')
# with a url
iris = pd.read_csv('https://tinyurl.com/iris-data')
```

Two ways of indexing

- `.loc[]` “label based”
- `.iloc[]` “position based”

For both you can use: a single value, a list of values, a boolean array. Two notable things:

1. If you use a slice notation with `.loc` (`'a':'f'`) the last value is included! (different from plain python and from `.iloc`)
2. Can be also a callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above)

Lecture XXII: More pandas

Group by

Data can be grouped with `groupby`, then *summary* function (`sum`, `mean`, ...) can be applied to each group at the same time.

```
iris = pd.read_csv('https://tinyurl.com/iris-data')

iris.groupby('variety').mean()
```

Groups are special lazy types which generate data only when needed for the summary operation.

Iterators

Object can be iterable. Python defines the iterator protocol as:

- `iterator.__iter__()` Return the iterator object itself. This is required to allow both containers and iterators to be used with the for and in statements.
- `iterator.__next__()` Return the next item from the container. If there are no further items, raise the StopIteration exception.

Notable iterators

Built-in lists, tuples, ranges, sets, dicts are iterators.

- Numpy arrays
- Pandas Series and DataFrames

Generators

```
def mygenerator() -> int:
    for i in [1, 6, 70, 2]:
        yield i
    print('Ended')  # Just to see when it reaches this
    ↪  point

g = mygenerator()

print(g)  # not useful
print(next(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))      # Exception
```

Pandas DataFrame

Be careful: the default iteration is on column names (similar to dicts, which iterate on keys).

- `iterrows()`: Iterate over the rows of a DataFrame as `(index, Series)` pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- `itertuples()`: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than `iterrows()`, and is in most cases preferable to use to iterate over the values of a DataFrame.

Iterating is slow: whenever possibile try to use vectorized operation or function application.

Pandas function application

```python
# apply the function to each column
df.apply(lambda col: col.mean() + 3)

# apply the function to each row
df.apply(lambda row: row + 3, axis=1)
```

Pandas query

```
df[df['A A'] > 3]

# equivalent to this (backticks because of the space)
df.query('`A A` > 3')

# query can also refer to the index
df.query('index >= 15')

# same as
df[15:]
```