WebScraping

Anno Accademico 2024-2025

Docente: Laura Ricci

Lezione 5

La libreria Pandas: Series

12 Febbraio 2025

Assignment 2: Dizionari e liste

- definire una lista contenente 5 dizionari: ogni dizionario contiene due coppie chiave-valore, nome ed età (in anni), di una persona
- utilizzare il meccanismo della list comprehension per produrre una lista di dizionari in cui ogni dizionario contiene 3 coppie chiave valore, il nome e l'età originari e l'**età in mesi**
- i dizionari della nuova lista dovranno includere solo le persone che **hanno più di 20 anni**

In [3]:

```
people = [ {"nome": "Sara", "età": 28}, {"nome": "Matteo", "età": 15},
           {"nome": "Emilia", "età": 6}, {"nome": "Maria", "età": 55},
           {"nome": "Giovanni", "età": 18} ]
people_month = [{"nome": x["nome"], "età": x["età"], "età_mesi": x["età"]*12}
                for x in people if x["età"] > 20]
people_month
```

Out[3]:

```
[{'nome': 'Sara', 'età': 28, 'età_mesi': 336},
 {'nome': 'Maria', 'età': 55, 'età_mesi': 660}]
```

Assignment 2: Dizionari e liste

In [4]:

```
def age_in_months(mylist):
    return [dict(one_person.items(), età_in_mesi=one_person['età'] * 12)
            for one_person in mylist
            if one_person['età'] > 20]

people_months = age_in_months(people)
people_months
```

Out[4]:

```
[{'nome': 'Sara', 'età': 28, 'età_in_mesi': 336},
 {'nome': 'Maria', 'età': 55, 'età_in_mesi': 660}]
```

Assignment 3: figli e nipoti

- Definire un dizionario che rappresenta i figli e nipoti appartenenti ad una famiglia: la chiave è il nome di un figlio F e il valore corrispondente è una lista di stringhe che rappresentano i figli di F
    - ad esempio il dizionario {'A':['B','C','D'], 'E':['F','G']} indica che 'A' ed 'E' sono fratelli, che i figli di 'A' sono 'B','C', e 'D' e che i figli di 'E' sono 'F' e 'G'.
- Utilizzare il meccanismo della list comprehension per creare una lista con i nomi di tutti i nipoti.

In [5]:

```
def grandchildren_names(d):
    return [one_grandchild
            for grandchild_list in d.values()
            for one_grandchild in grandchild_list]
```

Assignment 3: figli e nipoti

In [6]:

```
Dict={'Maria':['Paola', 'Anna', 'Giovanni'], 'Laura':['Matteo, Mario']}
grandchildren_names(Dict)
```

Out[6]:

```
['Paola', 'Anna', 'Giovanni', 'Matteo, Mario']
```

Assignment 4: vocali in stringhe

- è data una stringa contenente diverse parole seprate da uno spazio
- creare un dizionario in cui le chiavi siano le parole e i valori il numero di vocali in ogni parola
- ad esempio se la stringa è la seguente "this is an easy test" allora il dizionario risultante sarà {'this':1, 'is':1, 'an':1, 'easy':2, 'test':1}

In [8]:

```
def vowel_count(word):
    total = 0
    for one_letter in word.lower():
        if one_letter in 'aeiou':
            total += 1
    return total
```

Assignment 4: vocali in stringhe

In [9]:

```
def word_vowels(s):
    return {one_word: vowel_count(one_word)
            for one_word in s.split()}
word_vowels('this is an easy test')
```

Out[9]:

```
{'this': 1, 'is': 1, 'an': 1, 'easy': 2, 'test': 1}
```

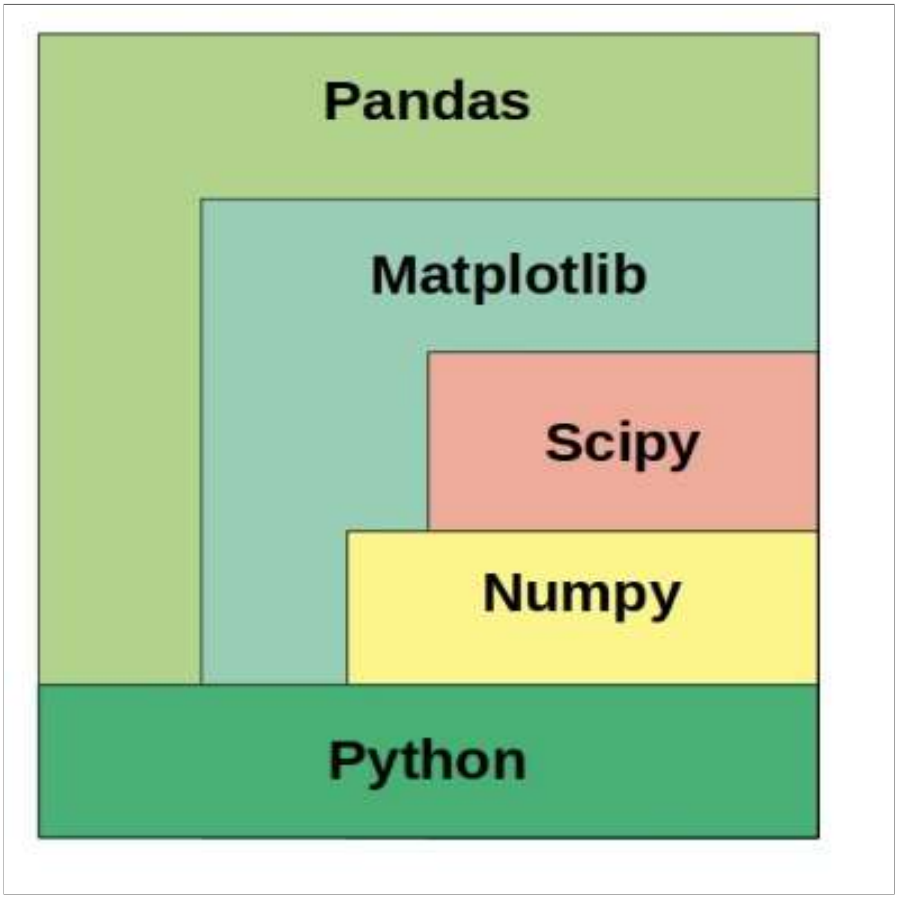
Pandas

- libreria open source per la **data analysis** costruita su **Python** e altre librerie
- utilizza routines C per la realizzazione efficace di molte funzionalità
- compatibile con **Excel** e **Google's sheets**
- come in questi framework, *Pandas* permette di interagire con dati rappresentati come *tabelle*, formate da righe e da colonne
- offre diverse funzionalità per lavorare su queste strutture
    - indexing
    - sorting,
    - filtering
    - cleaning
    - aggregating
    - pivoting
    - joining
    - deduping

Pandas

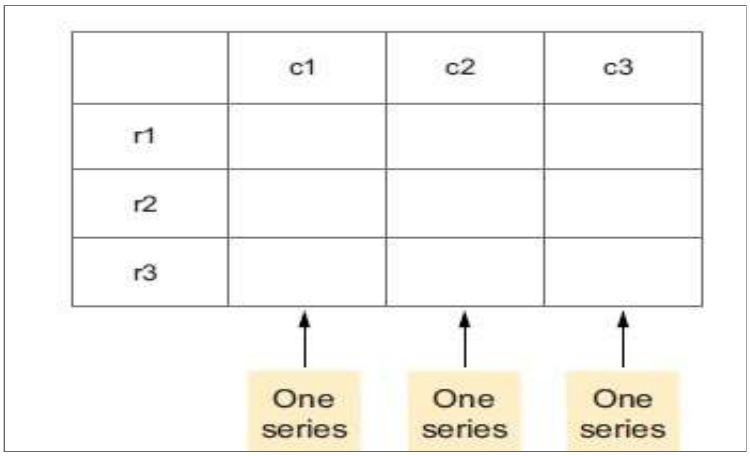
- offre anche la possibilità di *caricare* e *salvare* dati in formati tabulari "standard"
    - **CSV** (Comma-separated Values)
    - **TSV** (Tab-seprated values)
    - File **Excel**
    - Formati database *...
- operazioni numeriche e statistiche efficienti
- primitive di visualizzazione applicabili direttamente a strutture *Pandas*

Pandas: la gerarchia dei moduli



Pandas: Tipi di dato

- due tipi di dati principali
  - le **Series** rappresentano dati *1D*: ottime per rappresentare sequenze temporali
  - i **Data Frame** reppresentano dati *2D* come un file csv, una tabella di un database, ...
- quale è la relazione tra i due?
  - ogni colonna di un *DataFrame* è una *Series*
- lavoreremo con entrambe le strutture esaminando data-set reali



Le Series

- struttura monodimensionale
    - una sequenza **ordinata** di valori
    - dati omogenei *è possibile assegnare ad ogni valore contenuto in una **Series**
    - una label, ovvero un **label index** utilizzato per localizzare un elemento
    - la **posizione** dell'elemento nella serie (il primo elemento ha indice 0), ovvero un **position index**
- quale e' l'idea di base?
    - combinare caratteristiche di **liste** e di **dizionari**
    - come in una lista, gli elementi sono accessibili per indice
    - come in un dizionario si può accedere ad un elemento usando la label

Le Series

In [10]:

```
import numpy as np
import pandas as pd
from pandas import Series
Series()
```

Out[10]:

```
Series([], dtype: object)
```

- **pd.Series** è una classe della libreria Pandas
- invocando **pd.Series()** invoco il costruttore e creo un oggetto di quella classe, vuoto
- posso creare una Serie partendo da qualsiasi iterable, utilizzando come parametro del costruttore
- poichè non ho inserito elementi, Pandas per default assegna un tipo degli elementi uguale a **object**

Le Series: creazione a partire da iterable

- creare una serie di 10 elementi.
- ogni elemento contiene la valutazione, in trentesimi, di una prova sostenuta mensilmente da uno studente, in ogni mese tra **Settembre** e **Giugno**
- i valori contenuti nella serie devono essere valori casuali compresi tra **15 e 30**

In [11]:

```
g = np.random.default_rng(0)
s = Series(g.integers(15,30,10))
s
```

Out[11]:

```
0    27
1    24
2    22
3    19
4    19
5    15
6    16
7    15
8    17
9    27
dtype: int64
```

Le Series: parametri del costruttore

- per ottenere informazioni sui parametri del costruttore: premere **Shift + TAB**
- viene visualizzato ogni parametro insieme ad un valore **di default**, utilizzato se non si formisce alcun valore del parametro attuale
- ogni parametro che ha un valore di default, è, per definizione, **opzionale**

Le Series: indicizzazione

- ogni elemento della Series è identificato da un **indice numerico**, che è l'indice di default

In [12]:

```
s.index
```

Out[12]:

```
RangeIndex(start=0, stop=10, step=1)
```

- ma è anche possibile settare esplicitamente un **label index**, ad esempio indicizzare la serie rispetto ai mesi in cui sono avvenute le valutazioni

In [13]:

```
s.index='Set Ott Nov  Dic Gen Feb Mar Apr Mag Giu'.split()
s
```

Out[13]:

```
Set    27
Ott    24
Nov    22
Dic    19
Gen    19
Feb    15
Mar    16
Apr    15
Mag    17
Giu    27
dtype: int64
```

Le Series: indicizzazione

- possibile anche specificare il **label index** direttamente nel costruttore

In [14]:

```
months='Set Ott Nov  Dic Gen Feb Mar Apr Mag Giu'.split()
s=Series(g.integers(15,30,10), index=months)
s
```

Out[14]:

```
Set    24
Ott    28
Nov    22
Dic    24
Gen    29
Feb    25
Mar    24
Apr    23
Mag    23
Giu    29
dtype: int64
```

Series:"under the wood"

- *Pandas* delega la memorizzazione dei **valori** contenuti in una Series a **NumPy**
    - utilizza *ndarray*: struttura dati di *NumPy* usata per rappresentare matrici (a sua volta utilizza routine C)
    - scopo: utilizzare le operazioni vettoriali di *NumPy*, molto efficienti
        - *Series* come wrapper di oggetti *NumPy*
    - invece, gli attributi degli indici sono memorizzati in un oggetto **index** di **Pandas**
- una *Series* è quindi un oggetto che ne contiene altri due

Series: ”under the wood”

- verifichiamo che la series memorizza i suoi valori in un array **NumPy**
- gli attributi possono essere utilizzati per analizzare come è strutturato un oggetto
- analizziamo gli attributi della serie e vediamo come l'indice sia un tipo definito da **Pandas**

In [15]:

```
print(s.values)
type(s.values)
```

```
[24 28 22 24 29 25 24 23 23 29]
```

Out[15]:

```
numpy.ndarray
```

In [16]:

```
print(s.index)
type(s.index)
```

```
Index(['Set', 'Ott', 'Nov', 'Dic', 'Gen', 'Feb', 'Mar', 'Apr', 'Mag',
'Giu'], dtype='object')
```

Out[16]:

```
pandas.core.indexes.base.Index
```

Series: creazione a partire da oggetti Python

- una serie può essere creata a partire da oggetti *Python*, in maniera diretta o indiretta.
- **dizionari**:
    - per ogni chiave del dizionario, una corrispondente label della serie
- **tuple**
    - indice posizionale e label coincidono
- **array NumPy**
- un **set** non definisce un ordinamento: occorre convertirlo in una struttura dati che definisce un ordinamento

Series: data selection dictionary-like

In [17]:

```
import pandas as pd
population_dict = {'India': 1366417754,
                   'India': 1466428893,
                   'China': 1397715000,
                   'USA': 328239523,
                   'England': 55977200,
                   'Russia': 143666931,
                   'Japan':126264931}
population = Series(population_dict)
print(population)
```

```
India      1466428893
China      1397715000
USA         328239523
England      55977200
Russia      143666931
Japan       126264931
dtype: int64
```

Series: statistiche

In [18]:

```
print(f'Media delle valutazioni: {s.mean()}')
```

```
Media delle valutazioni: 25.1
```

- diversi tipi di aggregazioni
  - somma
  - media
  - mediana
  - deviazione standard
  - percentili

Series: indexers, loc e iloc

In [19]:

```
Regions = Series(['Toscana', 'Sicilia', 'Puglia', 'Sardegna'],
                 index=[2,4,6,8])
print(Regions[2])
```

```
Toscana
```

In [20]:

```
print(Regions[1])
```

```
---------------------------------------------------------------------
----
KeyError                                  Traceback (most recent call l
ast)
File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.13_qbz5
n2kfra8p0\LocalCache\local-packages\Python313\site-packages\pandas\core
\indexes\base.py:3805, in Index.get_loc(self, key)
   3804 try:
-> 3805     return self._engine.get_loc(casted_key)
   3806 except KeyError as err:

File index.pyx:167, in pandas._libs.index.IndexEngine.get_loc()

File index.pyx:196, in pandas._libs.index.IndexEngine.get_loc()

File pandas\\_libs\\hashtable_class_helper.pxi:2606, in pandas._libs.ha
shtable.Int64HashTable.get_item()

File pandas\\_libs\\hashtable_class_helper.pxi:2630, in pandas._libs.ha
shtable.Int64HashTable.get_item()

KeyError: 1

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call l
ast)
Cell In[20], line 1
----> 1 print(Regions[1])

File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.13_qbz5
n2kfra8p0\LocalCache\local-packages\Python313\site-packages\pandas\core
\series.py:1121, in Series.__getitem__(self, key)
   1118     return self._values[key]
   1120 elif key_is_scalar:
-> 1121     return self._get_value(key)
   1123 # Convert generator to list before going through hashable part
   1124 # (We will iterate through the generator there to check for sli
ces)
   1125 if is_iterator(key):

File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.13_qbz5
n2kfra8p0\LocalCache\local-packages\Python313\site-packages\pandas\core
\series.py:1237, in Series._get_value(self, label, takeable)
   1234     return self._values[label]
   1236 # Similar to Index.get_value, but we do not fall back to positi
onal
-> 1237 loc = self.index.get_loc(label)
   1239 if is_integer(loc):
   1240     return self._values[loc]

File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.13_qbz5
n2kfra8p0\LocalCache\local-packages\Python313\site-packages\pandas\core
\indexes\base.py:3812, in Index.get_loc(self, key)
   3807     if isinstance(casted_key, slice) or (
   3808         isinstance(casted_key, abc.Iterable)
   3809         and any(isinstance(x, slice) for x in casted_key)
```

```
   3810     ):
   3811         raise InvalidIndexError(key)
-> 3812     raise KeyError(key) from err
   3813 except TypeError:
   3814     # If we have a listlike key, _check_indexing_error will rai
se
   3815     #  InvalidIndexError. Otherwise we fall through and re-rais
e
   3816     #  the TypeError.
   3817     self._check_indexing_error(key)

KeyError: 1
```

Series: indexers, loc e iloc

- per evitare ambigità Pandas introduce due *indexers attribute* che indicano esplicitamente quale tipo di access si vuole utilizzare
  - accesso per label: **loc**
  - accesso diretto **iloc**

In [21]:

```
Regions = Series(['Toscana', 'Sicilia', 'Puglia', 'Sardegna'],  index=['2','4','6','8'])
print(Regions.loc['2'])
```

```
Toscana
```

In [22]:

```
Regions = Series(['Toscana', 'Sicilia', 'Puglia', 'Sardegna'],  index=['2','4','6','8'])
print(Regions.iloc[2])
```

```
Puglia
```

Series: slicing

- supponiamo di voler calcolare la media delle valutazioni nei primi 5 mesi e nei secondi 5 mesi
- possiamo utlizzare il meccanismo dello **slicing** anche sulle serie e sia per gli inidici numerici che per gli indici posizionali

In [23]:

```
s.iloc[:5].mean()
```

Out[23]:

```
np.float64(25.4)
```

In [24]:

```
s.iloc[5:].mean()
```

Out[24]:

```
np.float64(24.8)
```

In [25]:

```
s.loc['Set':'Gen'].mean()
```

Out[25]:

```
np.float64(25.4)
```

Series: indici non univoci

- creazione di una serie con label ripetute
- considereremo le temperature di febbraio (non bisestile) e indicizzeremo la serie con i giorni della settimana (ripetuti)

In [27]:

```
g = np.random.default_rng(0)
days='Dom Lun Mar Mer Gio Ven Sab'.split()
s= Series(g.normal(20,5,28),index=days*4)
s
```

Out[27]:

```
Dom    20.628651
Lun    19.339476
Mar    23.202113
Mer    20.524501
Gio    17.321653
Ven    21.807975
Sab    26.520000
Dom    24.735405
Lun    16.481324
Mar    13.672893
Mer    16.883628
Gio    20.206630
Ven     8.374846
Sab    18.906042
Dom    13.770445
Lun    16.338663
Mar    17.278705
Mer    18.418499
Gio    22.058153
Ven    25.212567
Sab    19.357327
Dom    26.832317
Lun    16.674027
Mar    21.757550
Mer    24.517351
Gio    20.470061
Ven    16.282504
Sab    15.391373
dtype: float64
```

Series: indici non univoci

In [28]:

```
s.loc['Lun']
```

Out[28]:

```
Lun    19.339476
Lun    16.481324
Lun    16.338663
Lun    16.674027
dtype: float64
```

In [29]:

```
s.loc['Lun'].mean()
```

Out[29]:

```
np.float64(17.20837234088955)
```

- è possibile anche selezionare alcuni elementi di una serie indicando una lista di indici

In [30]:

```
s = Series([10, 20, 30, 40, 50])
s.iloc[[2,4]]
```

Out[30]:

```
2    30
4    50
dtype: int64
```

Series: mask index, selezionare valori con espressioni booleane

- selezione di righe mediante lista di booleani
- alternativa al classico **for** + **espressione booleana**

In [31]:

```
s = Series([10, 20, 30, 40, 50])
s.iloc[[True, True, False, False, True]]
```

Out[31]:

```
0    10
1    20
4    50
dtype: int64
```

Series: mask index, selezionare valori con espressioni booleane

In [32]:

```
s.loc[s<30]
```

Out[32]:

```
0    10
1    20
dtype: int64
```

- l'espressione tra parentesi quadre restituisce una **serie di booleani**
- la serie di booleani viene quindi applicata alla serie s per selezionare gli elementi

Series: mask index, selezionare valori con espressioni booleane

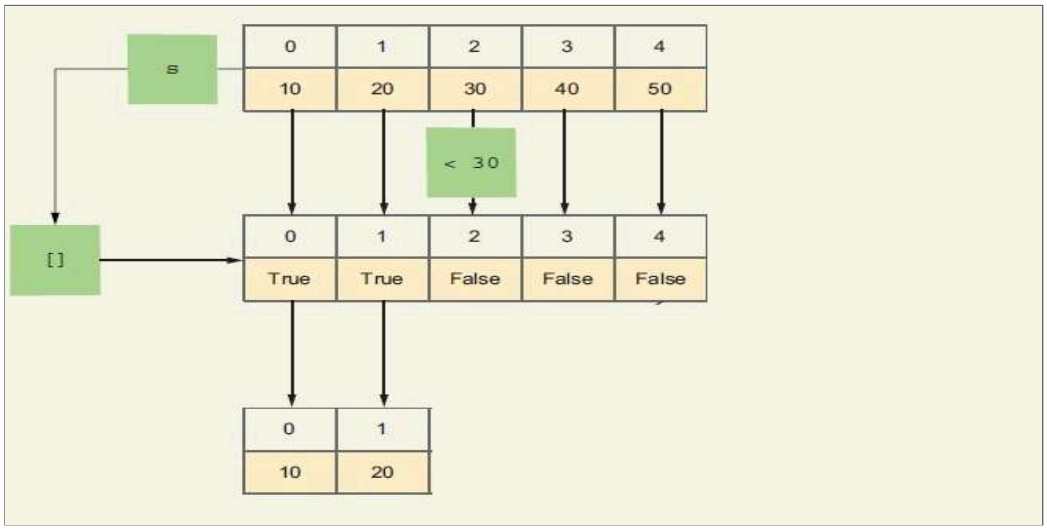
In [33]:

```
s.loc[s<=s.mean()]
```

Out[33]:

```
0    10
1    20
2    30
dtype: int64
```

- la serie s appare 3 volte nella espressione
  - calcolo della media
  - broadcast operation: confronto di ogni valore con la media
  - applicazione della serie booleana risultante a s



Series: dtype

- indica il tipo degli oggetti contenuti nella serie
- molti tipi di dato sono implementati da **NumPy** ed utilizzati da Pandas, ma esistono anche tipi di dato definiti da **Pandas**
- NumPy **dtype**:
    - **Integers** di diverse dimensioni **np.int8**, **np.int16**, **np.int32**, **np.int64**
    - **Unsigned** integers di diverse dimensioni **np.uint8**, **np.uint16**, **np.uint32**, **np.uint64**
    - **Floats** of diverse dimensioni **np.float16**, **np.float32**, **np.float64**.
    - **Objects** oggetti Python
- se si crea una serie
    - se tutti i valori sono interi, **dtype** è impostato a **np.int64**
    - se almeno un valore è un float, **dtype** è impostato a **np.float**
    - altrimenti, **dtype** è impostato a **object**

Series: dtype overriding

In [34]:

```
s = Series([10, 20, 30], dtype=np.float32)
s
```

Out[34]:

```
0    10.0
1    20.0
2    30.0
dtype: float32
```

- importante scegliere il tipo più adatto specialmente quando si lavora con datset grandi

In [35]:

```
s = s.astype(np.int64)
s.dtype
```

Out[35]:

```
dtype('int64')
```

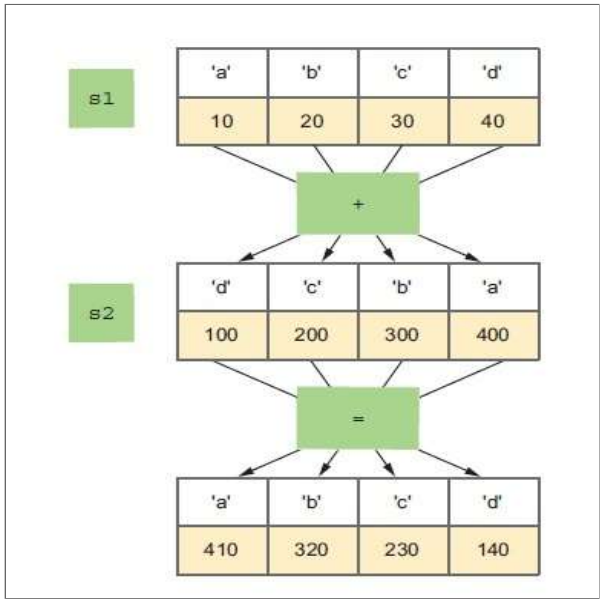
Series: vectorized operations

In [36]:

```
s1 = Series([10,20,30,40],index=list('abcd'))
s2 = Series([100,200,300,400],index=list('dcba'))
s1+s2
```

Out[36]:

```
a    410
b    320
c    230
d    140
dtype: int64
```



Series: broadcast

- operazione su una serie e su uno scalare
- applica lo scalare ad ogni elemento della serie
- mantiene tutti gli indici della serie di partenza
- +10 a tutte le valutazioni dello studente

In [37]:

```
s+10
```

Out[37]:

```
0    20
1    30
2    40
dtype: int64
```

Series: scalare le valutazioni, aggregazione e broadcast

- supponiamo la media M delle valutazioni di tutti gli studenti sia stata 25
- calcolare la differenza tra le valutazioni dello studente e M e aggiungerla ad ogni valutazione
- nella stessa funzione **aggregazione** e **broadcast**

In [38]:

```
g = np.random.default_rng(0)
s = Series(g.integers(18,30,10))
s+(25-s.mean())
```

Out[38]:

```
0    31.0
1    28.0
2    27.0
3    24.0
4    24.0
5    21.0
6    21.0
7    21.0
8    23.0
9    30.0
dtype: float64
```

Importare un data set in una Series da un file CSV

- **csv**: un formato semplice, testuale, utilizzato per scambiare dati
- plain-text usato per rappresentare dati tabulari
- la prima riga contiene i nomi delle colonne della tebella
- ogni riga contiene un numero finito di elementi che corrispondono alle diverse colonne
- ogni elemento separato dal successivo da una virgola
- può essere creato a partire da un file **Excel**

```
Title,Author,ISBN13,Pages
1984,George Orwell,978-0451524935,268
Animal Farm,George Orwell,978-0451526342,144
Brave New World,Aldous Huxley,978-0060929879,288
Fahrenheit 451,Ray Bradbury,978-0345342966,208
Jane Eyre,Charlotte Brontë,978-0142437209,532
Wuthering Heights,Emily Brontë,978-0141439556,416
Agnes Grey,Anne Brontë,978-1593083236,256
Walden,Henry David Thoreau,978-1420922615,156
Walden Two,B. F. Skinner,978-0872207783,301
"Eats, Shoots & Leaves",Lynne Truss,978-1592400874,209
```

La prima analisi di un dataset reale: corse di taxi a New York

- un dataset contenente diverse informazioni sulle corse dei taxi a New York
- il primo dataset contiene, per ogni corsa, i passeggeri

In [39]:

```
s=pd.read_csv('DataSets/taxi-passenger-count.csv',header=None).squeeze()
```

- la lettura di un file *csv* restituisce sempre un **DataFrame**
- se il dataset contiene una sola colonna, trasformarlo in una Series, utlilizzando **squeeze**
- la prima riga non contiene i nomi di colonna **header==None**

La prima analisi di un dataset reale: corse di taxi a New York

In [40]:

```
s.value_counts()
```

Out[40]:

```
0
1    7207
2    1313
5     520
3     406
6     369
4     182
0       2
Name: count, dtype: int64
```

- **value_counts()** restituisce una **Series** in cui i label indexes sono i valori distinti in **s** e i corrispondenti valori rappresentano la frequenza dil valore dell'indice nel dataset
- i valori sono ordinati

In [41]:

```
s.value_counts()[[3,4]]
```

Out[41]:

```
0
3    406
4    182
Name: count, dtype: int64
```

La seconda analisi di un dataset reale: categorizzazione dei dati

In [42]:

```
import pandas as pd
s=pd.read_csv('DataSets/taxi-distance.csv',header=None).squeeze()
s
```

Out[42]:

```
0       1.63
1       0.46
2       0.87
3       2.13
4       1.40
        ...
9994    2.70
9995    4.50
9996    5.59
9997    1.54
9998    5.80
Name: 0, Length: 9999, dtype: float64
```

- il secondo dataset contiene la distanza percorsa da ogni corsa.
- categorizzare i valori numerici in tre categorie di viaggi
    - **short**, < 2 km
    - **medi**, >= 2 km, ma <10km
    - **lunghi**, > 10 km

La seconda analisi di un dataset reale: categorizzazione dei dati

In [43]:

```
categories =s.astype(str)
categories.loc[:]='medium'
categories.loc[s<2]='short'
categories.loc[s>10]='long'
categories
```

Out[43]:

```
0        short
1        short
2        short
3       medium
4        short
         ...
9994    medium
9995    medium
9996    medium
9997     short
9998    medium
Name: 0, Length: 9999, dtype: object
```

Modifica di una serie

- in generale un metodo che modifica una serie ne costruiscono una nuova

In [44]:

```
s.head(3)
```

Out[44]:

```
0    1.63
1    0.46
2    0.87
Name: 0, dtype: float64
```

In [45]:

```
s.sort_values().head(3)
```

Out[45]:

```
6860    0.0
1605    0.0
2792    0.0
Name: 0, dtype: float64
```

In [46]:

```
s.head(3)
```

Out[46]:

```
0    1.63
1    0.46
2    0.87
Name: 0, dtype: float64
```

Un DataSet di temperature minime

- descrive le temperature minime della città di Melbourne (Australia)
- considerati **10 anni (1981-1990)**
- rilevazioni giornaliere
- sorgente dei dati: *Australian Bureau of Meteorology*
- file: **daily-min-temperatures.csv**

```
"Date","Temp"
"1981-01-01",20.7
"1981-01-02",17.9
"1981-01-03",18.8
"1981-01-04",14.6
"1981-01-05",15.8
"1981-01-06",15.8
"1981-01-07",15.8
"1981-01-08",17.4
"1981-01-09",21.8
"1981-01-10",20.0
"1981-01-11",16.2
"1981-01-12",13.3
"1981-01-13",16.7
"1981-01-14",21.5
"1981-01-15",25.0
"1981-01-16",20.7
"1981-01-17",20.6
"1981-01-18",24.8
```

Importare un DataSet: read_csv importa in un DataFrame

- per default, **Pandas** ricerca il dataset nella stessa directory del Notebook

In [47]:

```
from pandas import read_csv
from matplotlib import pyplot
DF = read_csv('DataSets/daily-min-temperatures.csv', parse_dates=True)
DF
```

Out[47]:

	Date	Temp
0	1981-01-01	20.7
1	1981-01-02	17.9
2	1981-01-03	18.8
3	1981-01-04	14.6
4	1981-01-05	15.8
...	...	...
3645	1990-12-27	14.0
3646	1990-12-28	13.6
3647	1990-12-29	13.5
3648	1990-12-30	15.7
3649	1990-12-31	13.0

3650 rows × 2 columns

to_datetime

In [48]:

```
import pandas as pd
StringSeries = Series(['10/25/2005','10/29/2002','01/01/2001'])
print(StringSeries)
DateSeries=pd.to_datetime(StringSeries)
print(DateSeries)
```

```
0    10/25/2005
1    10/29/2002
2    01/01/2001
dtype: object
0   2005-10-25
1   2002-10-29
2   2001-01-01
dtype: datetime64[ns]
```

to_datetime

In [49]:

```
# create a datetime string
date_string = '2001-12-24 12:38'
print("String:", date_string)
# convert string to datetime
date = pd.to_datetime(date_string)
print("DateTime:", date)
```

```
String: 2001-12-24 12:38
DateTime: 2001-12-24 12:38:00
```

Lavorare con datetime

- diversi usi
    - time series analysis
    - data filtering in base a specifici intervalli di tempo
    - calcolare differenza tra due date
    - raggruppare in base a mese, giorno,...
- lo utilizzeremo spesso nel corso

Lavorare con datetime: strftime

In [50]:

```
from datetime import datetime
now = datetime.now() # current date and time
year = now.strftime("%Y")
print("year:", year)
month = now.strftime("%m")
print("month:", month)
day = now.strftime("%d")
print("day:", day)
time = now.strftime("%H:%M:%S")
print("time:", time)
date_time = now.strftime("%m/%d/%Y, %H:%M:%S")
print("date and time:",date_time)
print(type(date_time))
```

```
year: 2025
month: 02
day: 14
time: 09:57:35
date and time: 02/14/2025, 09:57:35
<class 'str'>
```

Importare un DataSet in una Series

In [51]:

```
from pandas import read_csv
from matplotlib import pyplot
temp_series = read_csv('DataSets/daily-min-temperatures.csv',
              index_col='Date', parse_dates=True).squeeze('columns')
temp_series
```

Out[51]:

```
Date
1981-01-01    20.7
1981-01-02    17.9
1981-01-03    18.8
1981-01-04    14.6
1981-01-05    15.8
              ...
1990-12-27    14.0
1990-12-28    13.6
1990-12-29    13.5
1990-12-30    15.7
1990-12-31    13.0
Name: Temp, Length: 3650, dtype: float64
```

- una delle colonne contenute nel file **csv** diventa il **label index**: la indico con **index_col**

Statistiche descrittive

In [52]:

```
temp_series.nunique()
```

Out[52]:

```
229
```

In [53]:

```
temp_series.value_counts(ascending=True)
```

Out[53]:

```
Temp
24.8     1
21.9     1
0.1      1
24.1     1
23.4     1
        ..
12.5    44
10.5    44
13.0    48
10.0    51
11.0    51
Name: count, Length: 229, dtype: int64
```

- restituisce una **Series** in cui l'**index label** sono i valori delle temperature e i valori sono le rispettive occorrenze nella serie
- ordinata rispetto alle occorrenze

Statistiche descrittive

In [54]:

```
temp_series.describe()
```

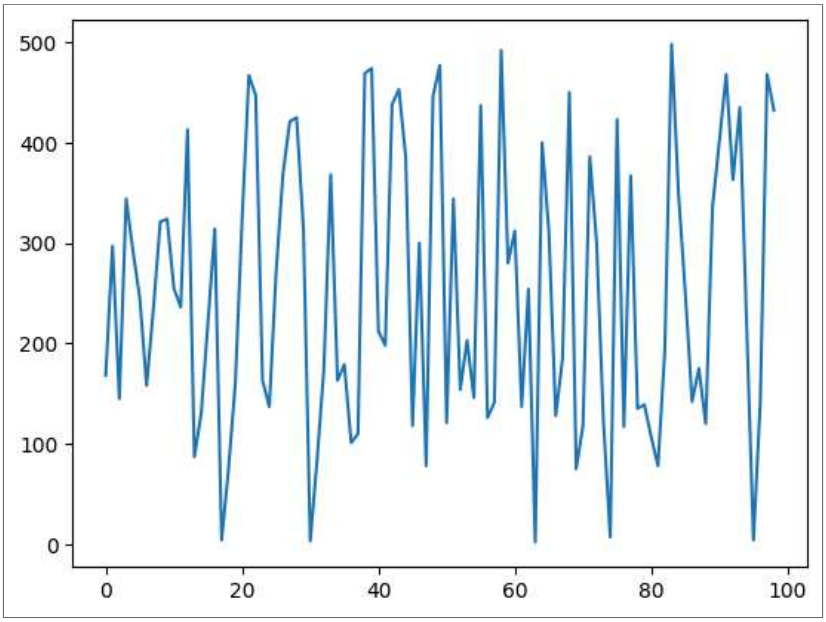
Out[54]:

```
count    3650.000000
mean       11.177753
std         4.071837
min         0.000000
25%         8.300000
50%        11.000000
75%        14.000000
max        26.300000
Name: Temp, dtype: float64
```

Visualizzare le Series

In [55]:

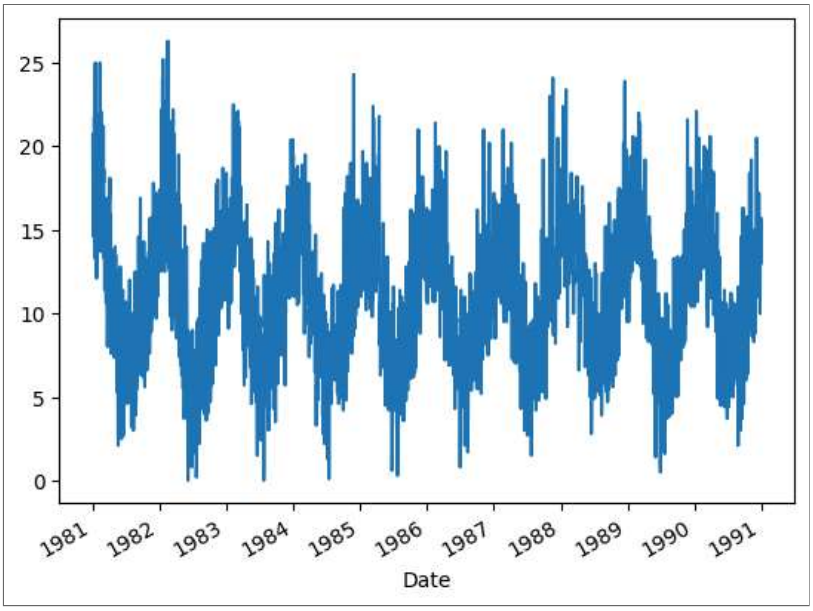
```
import numpy as np
s = pd.Series([np.random.randint(1,500) for i in range(1,100)])
p = pyplot.plot(s.index, s.values)
```



Visualizzare le temperature minime: default line plot

In [56]:

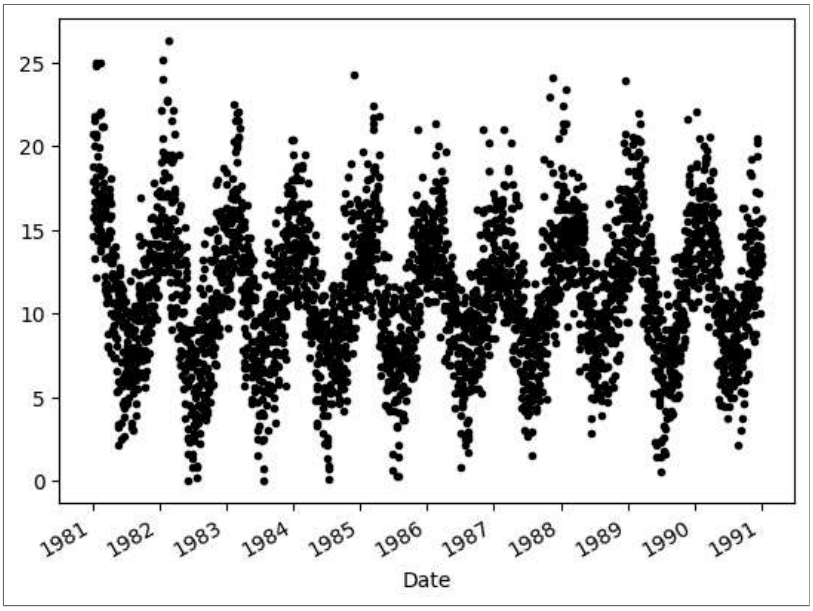
```
temp_series.plot()
pyplot.show()
```



Il DataSet delle temperature minime: visualizzazione

In [57]:

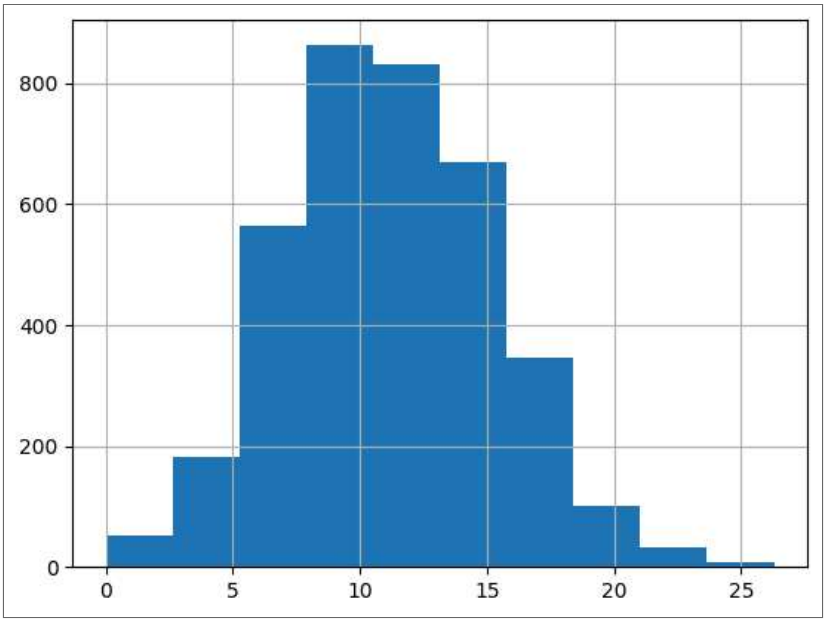
```
temp_series.plot(style='k.')
pyplot.show()
```



Il DataSet delle temperature minime:istogrammi

In [58]:

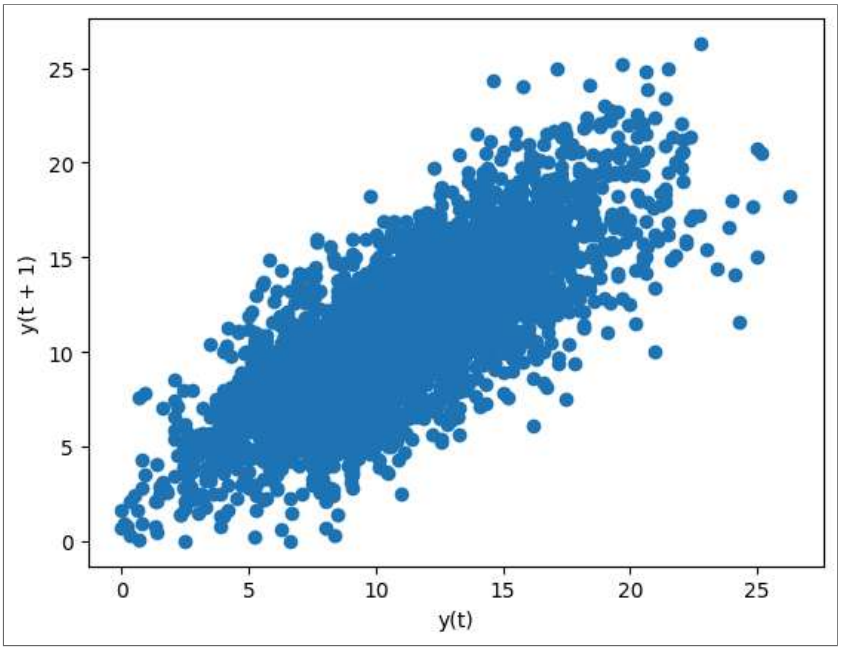
```
temp_series.hist()
pyplot.show()
```



Il DataSet delle temperature minime: lag plot

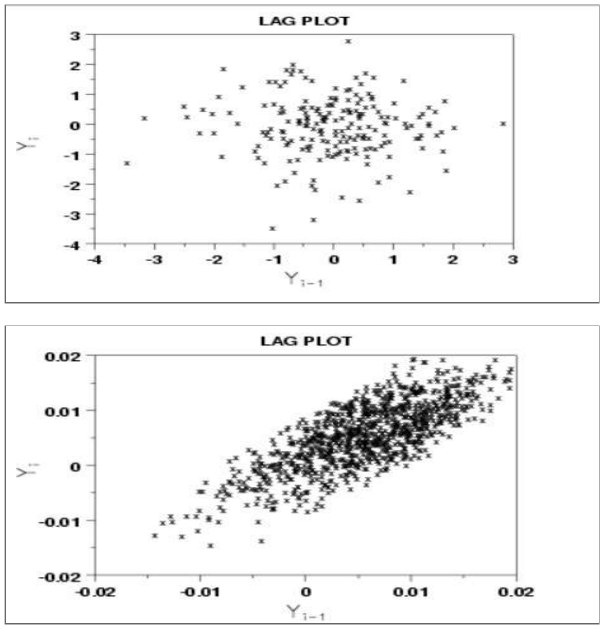
In [59]:

```
from pandas.plotting import lag_plot
lag_plot(temp_series)
pyplot.show()
```



Il DataSet delle temperature minime: lag_plot

- visualizza se i valori di una time-series siano random o meno
- dati random non mostrano una struttura identificabile nel lag-plot
- dati non-random mostrano una struttura identificabile



Il DataSet delle temperature minime: confronto tra anni

In [60]:

```
from pandas import Grouper
import matplotlib.pyplot as plt
groups = temp_series.groupby(Grouper(freq='YE'))
for name, group in groups:
    print(name)
    print(group)
```

```
1981-12-31 00:00:00
Date
1981-01-01    20.7
1981-01-02    17.9
1981-01-03    18.8
1981-01-04    14.6
1981-01-05    15.8
              ...
1981-12-27    15.5
1981-12-28    13.3
1981-12-29    15.6
1981-12-30    15.2
1981-12-31    17.4
Name: Temp, Length: 365, dtype: float64
1982-12-31 00:00:00
Date
1982-01-01    17.0
1982-01-02    15.0
1982-01-03    13.5
1982-01-04    15.2
1982-01-05    13.0
              ...
1982-12-27    15.3
1982-12-28    16.3
1982-12-29    15.8
1982-12-30    17.7
1982-12-31    16.3
Name: Temp, Length: 365, dtype: float64
1983-12-31 00:00:00
Date
1983-01-01    18.4
1983-01-02    15.0
1983-01-03    10.9
1983-01-04    11.4
1983-01-05    14.8
              ...
1983-12-27    13.9
1983-12-28    11.1
1983-12-29    16.1
1983-12-30    20.4
1983-12-31    18.0
Name: Temp, Length: 365, dtype: float64
1984-12-31 00:00:00
Date
1984-01-01    19.5
1984-01-02    17.1
1984-01-03    17.1
1984-01-04    12.0
1984-01-05    11.0
              ...
1984-12-26    12.2
1984-12-27    12.0
1984-12-28    12.6
1984-12-29    16.0
1984-12-30    16.4
Name: Temp, Length: 365, dtype: float64
1985-12-31 00:00:00
```

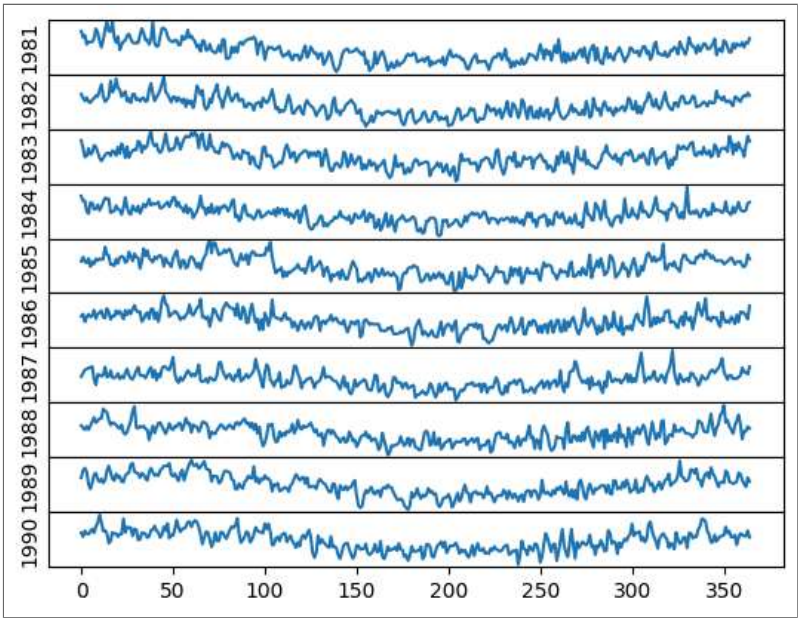
```
Date
1985-01-01    13.3
1985-01-02    15.2
1985-01-03    13.1
1985-01-04    12.7
1985-01-05    14.6
              ...
1985-12-27    11.5
1985-12-28    10.8
1985-12-29    12.0
1985-12-30    16.3
1985-12-31    14.4
Name: Temp, Length: 365, dtype: float64
1986-12-31 00:00:00
Date
1986-01-01    12.9
1986-01-02    13.8
1986-01-03    10.6
1986-01-04    12.6
1986-01-05    13.7
              ...
1986-12-27    14.6
1986-12-28    14.2
1986-12-29    13.2
1986-12-30    11.7
1986-12-31    17.2
Name: Temp, Length: 365, dtype: float64
1987-12-31 00:00:00
Date
1987-01-01    12.3
1987-01-02    13.8
1987-01-03    15.3
1987-01-04    15.6
1987-01-05    16.2
              ...
1987-12-27    16.2
1987-12-28    14.2
1987-12-29    14.3
1987-12-30    13.3
1987-12-31    16.7
Name: Temp, Length: 365, dtype: float64
1988-12-31 00:00:00
Date
1988-01-01    15.3
1988-01-02    14.3
1988-01-03    13.5
1988-01-04    15.0
1988-01-05    13.6
              ...
1988-12-26     9.5
1988-12-27    12.9
1988-12-28    12.9
1988-12-29    14.8
1988-12-30    14.1
Name: Temp, Length: 365, dtype: float64
1989-12-31 00:00:00
Date
1989-01-01    14.3
```

```
1989-01-02    17.4
1989-01-03    18.5
1989-01-04    16.8
1989-01-05    11.5
              ...
1989-12-27    13.3
1989-12-28    11.7
1989-12-29    10.4
1989-12-30    14.4
1989-12-31    12.7
Name: Temp, Length: 365, dtype: float64
1990-12-31 00:00:00
Date
1990-01-01    14.8
1990-01-02    13.3
1990-01-03    15.6
1990-01-04    14.5
1990-01-05    14.3
              ...
1990-12-27    14.0
1990-12-28    13.6
1990-12-29    13.5
1990-12-30    15.7
1990-12-31    13.0
Name: Temp, Length: 365, dtype: float64
```

Il DataSet delle temperature minime: confronto tra anni

In [61]:

```
fig, axs =plt.subplots(nrows=10,ncols=1,sharex=True)
fig.subplots_adjust(hspace=0)
i=0
for name, group in groups:
    axs[i].plot(group.values)
    axs[i].set_yticks(())
    axs[i].set(ylabel=name.year)
    i=i+1
```



Il DataSet delle temperature minime: heatmap

- necessario rappresentare i dati come una matrice **NumPy**

In [62]:

```
import numpy as np
list=[]
for name, group in groups:
   groupnp=group.to_numpy()
   list.append(groupnp)
matrix=np.asmatrix(list)
print(matrix)
dimensions=np.shape(matrix)
print(dimensions)
```

```
 [[20.7 17.9 18.8 ... 15.6 15.2 17.4]
  [17.  15.  13.5 ... 15.8 17.7 16.3]
  [18.4 15.  10.9 ... 16.1 20.4 18. ]
  ...
  [15.3 14.3 13.5 ... 12.9 14.8 14.1]
  [14.3 17.4 18.5 ... 10.4 14.4 12.7]
  [14.8 13.3 15.6 ... 13.5 15.7 13. ]]
 (10, 365)
```

Il DataSet delle temperature minime: heatmap

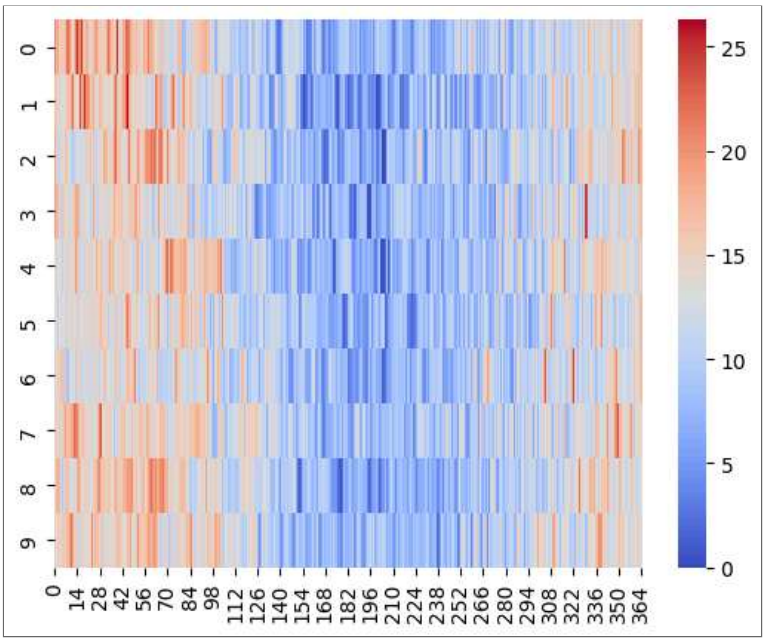
- visualizzare in una heatmap le rivazioni di temperatura
- asse x giorni, asse y anni, colore della map più freddo o caldo, a seconda della temperatura

In [63]:

```
import seaborn as sns
sns.heatmap(matrix, cmap='coolwarm')
```

Out[63]:

```
<Axes: >
```



Il DataSet delle temperature minime: anno 1990, confronto tra i mesi

In [64]:

```
one_year = temp_series['1990']
one_year_groups = one_year.groupby(Grouper(freq='ME'))
list_month=[]
for name, group in one_year_groups:
   months_np=group.to_numpy()
   for i in range(len(months_np),31):
        months_np= np.append(months_np,-1)
   list_month.append(months_np)
   matrix=np.asmatrix(list_month)
dimensions=np.shape(matrix)
print(dimensions)
```

```
(12, 31)
```

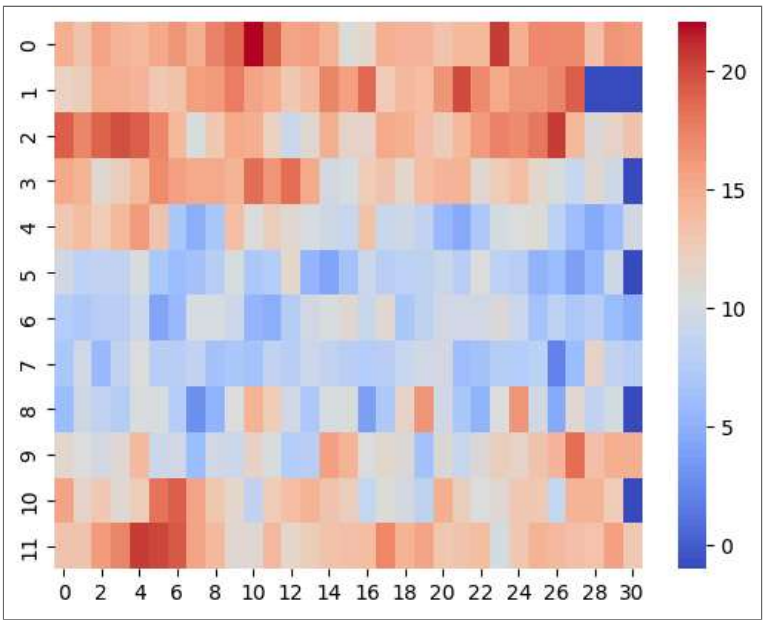
Il DataSet delle temperature minime: anno 1990, confronto tra i mesi

In [65]:

```
sns.heatmap(matrix, cmap='coolwarm')
```

Out[65]:

```
<Axes: >
```



Applicare una funzione ad ogni valore in una Series

- funzioni sono *first class objects* in Python
- possono essere passate come argomento di un'altra funzione
- le *Series* definiscono un metodo **apply**
    - riceve in input il nome di una funzione
    - applica il nome di quella funzione a tutti i valori della serie

Assignment 5

In [66]:

```
s = pd.Series([np.random.randint(1,10) for i in range(1,100)])
print(s)
s.apply(func='square')
```

```
0     2
1     8
2     7
3     9
4     6
     ..
94    1
95    2
96    5
97    4
98    9
Length: 99, dtype: int64
```

Out[66]:

```
0     4
1    64
2    49
3    81
4    36
     ..
94    1
95    4
96   25
97   16
98   81
Length: 99, dtype: int64
```

In [67]:

```
superheroes =[ "Batman","Superman","Spider.man","Iron-Man",
               "Captain America","Wonder Woman"]
strenght_levels = (100, 120, 90, 95, 110, 120)
```

- utilizzare le lista di superheroes per popolare un oggetto di tipo *Series*
- utilizzare la tupla di strenght_levels per popolare un oggetto di tipo *Series*

- creare una *Series* con i *superheroes* come index labels e *strength_level* come valore. Assegnare la *Series* alla variabile *heroes*
- estrarre i primi due elementi e gli ultimi due elementi della *Series heroes*
- determinare il numero di valori unici di *heroes*
- calcolare la forza media, la massima e la minima di *heroes*
- duplicare il livello di forza di ogni *superhero*
- convertire la *Series heroes* in un dizionario *Python*

Assignment 6

- E' dato il *dataset revolutionary_war.csv* che contiene diverse battaglie combattute durante la Rivoluzione Americana, con la descrizione del nome della battaglia, la data di inizio, lo stato in cui si è combattuta
- uno storico è interessato a conoscere in quale giorno della settimana sono iniziate il maggior numero di Battaglie durante la rivoluzione
- importare il dataset in una *series* trascurando le colonne che si riferiscono al nome della Battaglia e allo stato, e mantenendo solo la colonna che riguarda la data di inizio della battaglia
- lavorare sulla *series* risultante