Introduzione a Pandas

La libreria **Pandas** è lo strumento essenziale per la gestione e analisi dei dati in Python. Permette di lavorare con dati provenienti da sorgenti eterogenee come fogli Excel, file CSV, JSON e database SQL.

Strutture Dati Principali

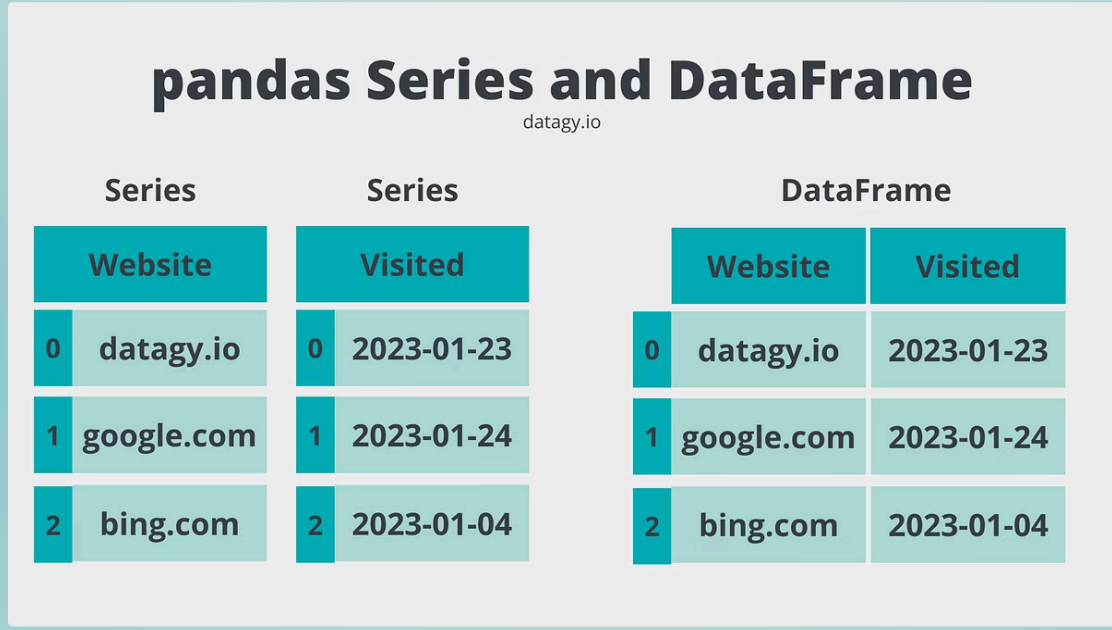
Series

Array **unidimensionale** con etichette personalizzate.

DataFrame

Tabella **bidimensionale** strutturata su **colonne**, dove ogni riga rappresenta un'osservazione

Entrambe supportano la presenza di dati eterogenei anche se è sconsigliato averne.



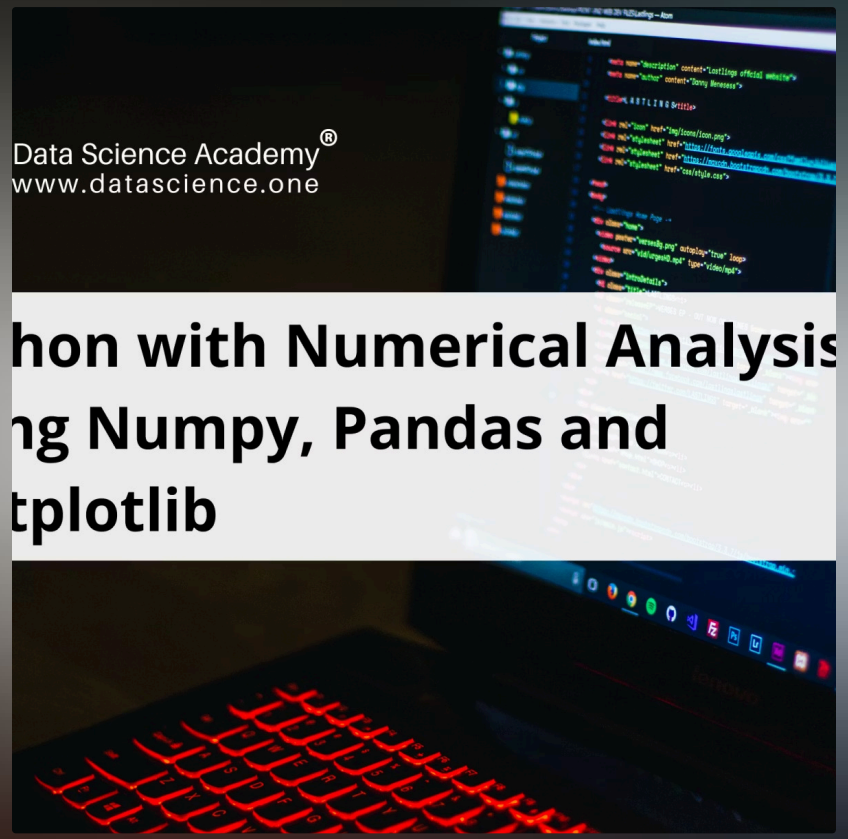
Creazione di una Series

Una Series è un **array unidimensionale** con etichette personalizzate chiamate "**index**". Supporta **etichette non numeriche** e gestisce automaticamente i dati mancanti.

```
grades = pd.Series([27, 30, 24, 18, 22, 20, 29])
print(grades)
```

Output:

```
0 27
1 30
2 24
3 18
4 22
5 20
6 29
dtype: int64
```



Caratteristiche delle Series

Una Series è una struttura **dati unidimensionale** (come una colonna di una tabella) dove ogni valore è associato a un'etichetta, chiamata "**indice**". Questo la rende molto versatile, anche per gestire dati temporali.

Le etichette di una Series **non devono essere necessariamente uniche** e possono essere di vario tipo (numeri, testo, date). Puoi accedere facilmente ai dati sia tramite la loro posizione numerica (partendo da zero, come in una lista) sia direttamente tramite l'etichetta assegnata.

Un grande vantaggio è che le Series **gestiscono automaticamente i dati mancanti** (spesso indicati come "NaN" - Not a Number), escludendoli dai calcoli statistici per risultati più precisi.

Quando esegui operazioni tra due Series (come somma, sottrazione, ecc.), queste vengono allineate automaticamente in base alle loro etichette. Non importa se le Series hanno lunghezze diverse; i valori vengono abbinati correttamente. L'indice della Series risultante sarà l'unione ordinata di tutti gli indici delle Series di partenza.

Indici Non Unici - Comportamento

Quando si accede a una Series di Pandas, il comportamento di recupero dei dati dipende dalla **natura degli indici.** È fondamentale comprendere come gli indici non unici vengano gestiti.

Ecco un esempio pratico che dimostra questo comportamento:

```
# Esempio con indici duplicati
voti = pd.Series([28, 30, 25], index=['Mario', 'Anna', 'Mario'])
print(voti['Mario'])
```

Output:

```
Mario    28
Mario    25
dtype: int64
```

Come si può notare dall'esempio, Pandas adotta un comportamento coerente e prevedibile:

- Se l'indice a cui si accede è **unico**, la Series restituisce un singolo valore (scalare).
- Se l'indice è **duplicato**, come nel caso di 'Mario', Pandas restituisce una nuova Series contenente **tutti i valori** corrispondenti a quell'indice.

Attenzione: Quando si lavora con Series che potrebbero avere indici duplicati, è buona pratica utilizzare esplicitamente l'indicizzatore `.loc[]`. Questo rende il codice più leggibile e intende chiaramente l'intenzione di accedere per etichetta.

Gestione dati mancanti

Rappresentazione Automatica

Pandas utilizza valori speciali come **NaN** (Not a Number) per rappresentare i dati mancanti, senza causare errori nel codice. Questo permette di lavorare con dataset incompleti senza interruzioni.

Gestione Trasparente nelle Operazioni

Quando esegui operazioni matematiche o statistiche, Pandas **gestisce automaticamente i valori mancanti:**

Le operazioni aritmetiche ignorano i NaN Le funzioni statistiche come **.mean()**, **.sum()** escludono automaticamente i valori mancanti Il conteggio con **.count() c**onsidera solo i valori validi Metodi Specifici per Dati Mancanti

Pandas offre strumenti dedicati:

.isnull() e **.notnull()** per identificare valori mancanti **.dropna()** per rimuovere righe/colonne con dati mancanti .**fillna()** per sostituire i valori mancanti con altri valori **.interpolate()** per stimare valori mancanti basandosi sui dati circostanti

Accesso agli Elementi

Indicizzazione Singola

```
grades[0]  # 27
```

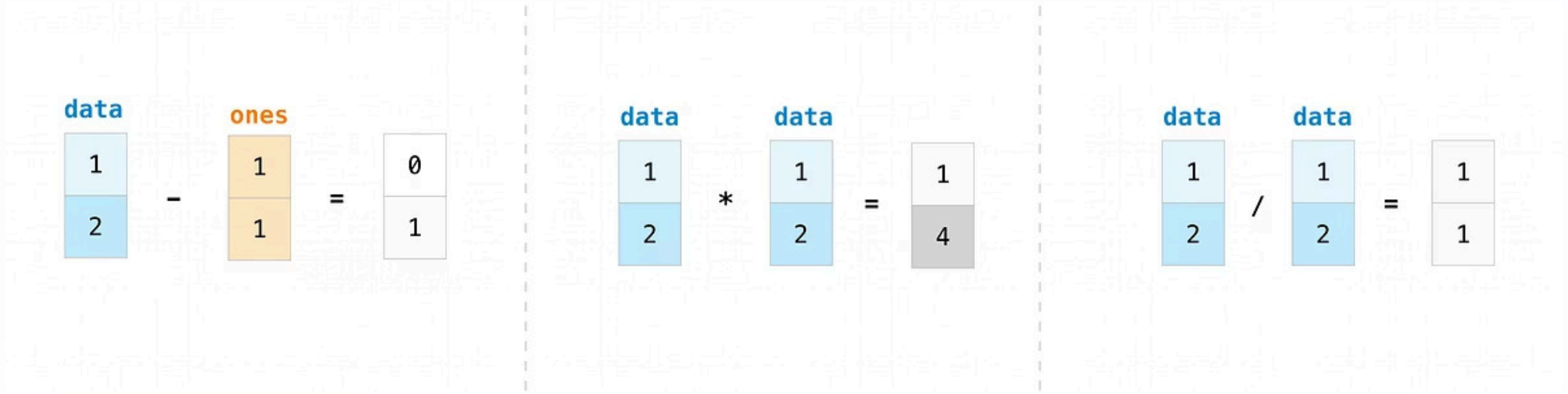
Slicing

```
grades[0:3]
# 0    27
# 1    30
# 2    24
```

Filtraggio Booleano

```
grades > 24
# 0    True
# 1    True
# 2    False
# ...
```

```
grades[grades > 24]
# 0 27
# 1 30
# 6 29
```



Operazioni Matematiche

Le Series supportano **operazioni aritmetiche** e **funzioni matematiche** applicate a tutti gli elementi:

Divisione

```
grades / 10
# 0   2.7
# 1   3.0
# 2   2.4
```

Radice Quadrata

```
np.sqrt(grades)
# 0 5.196152
# 1 5.477226
# 2 4.898979
```

Statistiche Descrittive

Le **Series** offrono metodi integrati per calcolare **statistiche descrittive**:

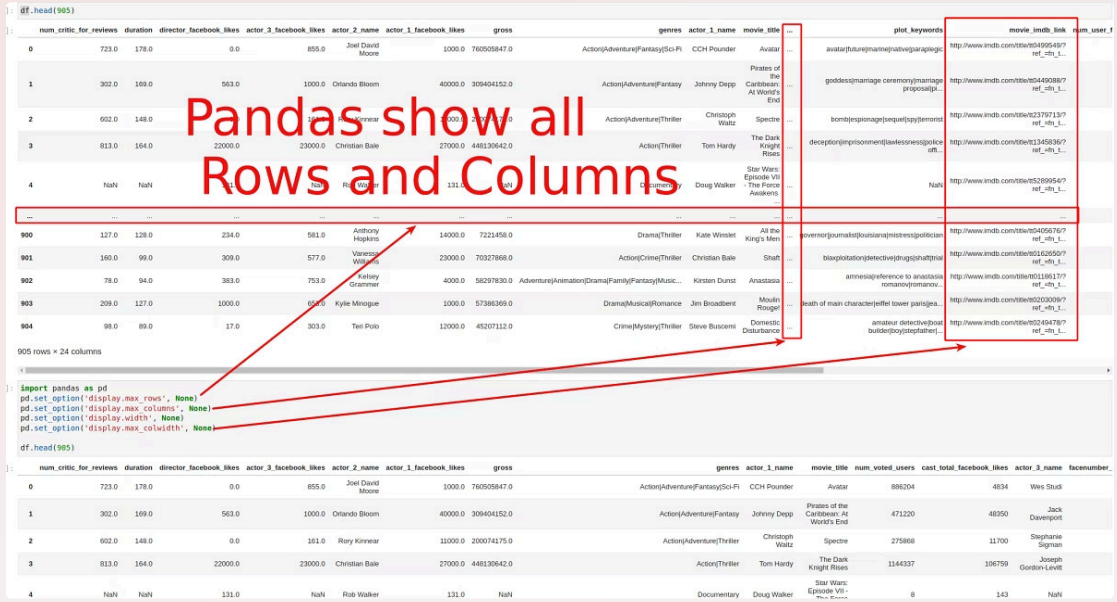
7	24.3	18	30
Count	**Media**	**Minimo**	**Massimo**
Numero di elementi	Valore medio	Valore più basso	Valore più alto

Il metodo `.describe()` fornisce un riepilogo completo di tutte le statistiche principali.

Creazione di DataFrame

Un **DataFrame** è una collezione di **Series** che formano una **tabella bidimensionale**. Può combinare diversi tipi di dati, **mantenendo la coerenza dei tipi** all'interno di ogni colonna.

```
data = {
 "name": ["Maria", "Anna", "Francesco", "Cristina"],
 "sex": ["f", "f", "m", "f"],
 "group": ["a", "b", "a", "b"],
 "x": [1, 2, 3, 4],
 "y": [8, 9, 10, 11]
}
frame = pd.DataFrame(data)
```



Metodi di Costruzione DataFrame

1

Da Dizionario

Il metodo più comune, usando dizionari con liste di uguale lunghezza

2

Colonna per Colonna

Creazione di DataFrame vuoto e aggiunta progressiva delle colonne

3

Da File Esterni

Importazione diretta da CSV, Excel, JSON o database

Lettura Dati da File

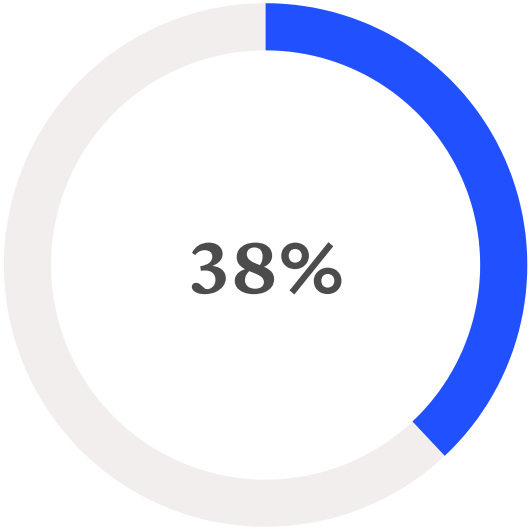
Pandas supporta la lettura di dati da molteplici formati. L'importazione è il primo passo fondamentale nell'analisi dei dati.

```
url = "https://raw.githubusercontent.com/pandas-dev/pandas/master/doc/data/titanic.csv"
titanic = pd.read_csv(url, index_col='Name')
```

Il metodo `.head()` visualizza le prime cinque righe per un'anteprima rapida dei dati importati.

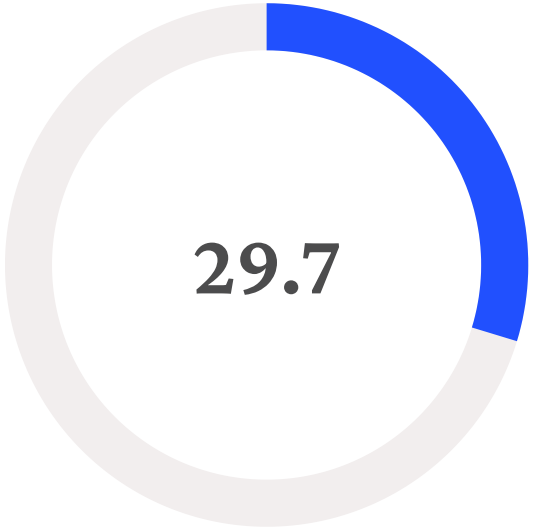
Dataset Titanic - Panoramica

Il dataset Titanic contiene informazioni sui passeggeri: sopravvivenza, classe, sesso, età, prezzo del biglietto. Le statistiche descrittive rivelano:



Sopravvivenza

Percentuale di passeggeri sopravvissuti

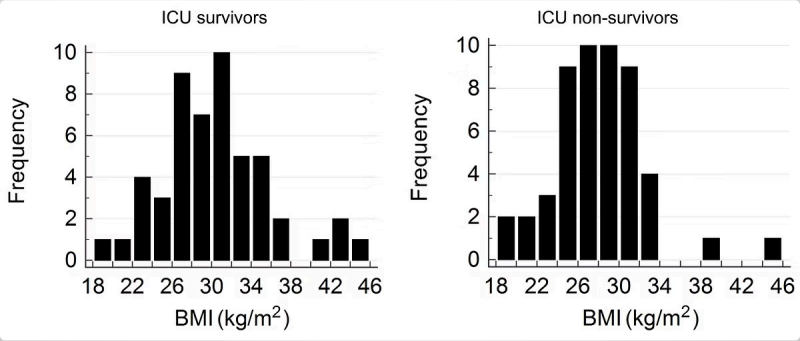


Età Media

Anni di età media dei passeggeri

Prezzo Max

Dollari per il biglietto più costoso



Analisi Sopravvivenza per Età

Con poche righe di codice possiamo analizzare la relazione tra età e sopravvivenza:

```
print(titanic.groupby("Survived")["Age"].mean())
# Survived
# 0    30.626179  (non sopravvissuti)
# 1    28.343690  (sopravvissuti)
```

I sopravvissuti erano in media più giovani di circa 2 anni rispetto ai non sopravvissuti. Gli istogrammi della distribuzione dell'età mostrano chiaramente questa differenza.

Funzione mean()

La funzione .mean() di Pandas è uno strumento fondamentale per l'analisi statistica, che consente di calcolare la **media aritmetica** di un set di dati. È estremamente versatile e può essere applicata sia a singole colonne (Series) che a intere tabelle (DataFrame), offrendo un modo rapido per ottenere una misura di tendenza centrale.

Cos'è e cosa calcola?

La funzione .mean() calcola la **media aritmetica**, ovvero la somma di tutti i valori numerici divisa per il numero totale di valori. È utile per capire il "valore tipico" o "centrale" di una distribuzione di numeri. In Pandas, può essere usata per trovare la media di età, stipendi, punteggi, ecc., all'interno dei tuoi dati.

Gestione dei valori mancanti (NaN)

Una delle caratteristiche importanti di .mean() in Pandas è la sua gestione automatica dei valori mancanti (Not a Number, NaN). Per impostazione predefinita, la funzione **ignora i valori** NaN nel calcolo, evitando che influenzino il risultato della media. Questo comportamento può essere controllato tramite il parametro skipna.

Esempi pratici

Vediamo come utilizzare .mean() su diversi tipi di oggetti Pandas.

Su una Series (colonna singola)

```
import pandas as pd
import numpy as np

# Creiamo una Series di esempio
eta = pd.Series([25, 30, 35, 40, np.nan, 20])
print("Series originale:")
print(eta)

# Calcoliamo la media dell'età
media_eta = eta.mean()
print(f"\nMedia dell'età: {media_eta:.2f}")

# Output previsto:
# Series originale:
# 0    25.0
# 1    30.0
# 2    35.0
# 3    40.0
# 4     NaN
# 5    20.0
# dtype: float64
#
# Media dell'età: 30.00 (somma dei valori presenti / numero dei valori presenti, NaN ignorato)
```

Su un DataFrame (tabelle)

```
# Creiamo un DataFrame di esempio
dati_prestazioni = pd.DataFrame({
    'Vendite_Q1': [100, 150, 120, np.nan, 130],
    'Vendite_Q2': [110, 160, 130, 140, 150],
    'Spese_Q1': [50, 70, 60, 55, 65]
})
print("DataFrame originale:")
print(dati_prestazioni)

# Calcoliamo la media per ciascuna colonna numerica
media_colonne = dati_prestazioni.mean()
print("\nMedia per ciascuna colonna:")
print(media_colonne)

# Output previsto:
# DataFrame originale:
#    Vendite_Q1  Vendite_Q2  Spese_Q1
# 0       100.0         110        50
# 1       150.0         160        70
# 2       120.0         130        60
# 3         NaN         140        55
# 4       130.0         150        65
#
# Media per ciascuna colonna:
# Vendite_Q1    125.0
# Vendite_Q2    138.0
# Spese_Q1       60.0
# dtype: float64
```

Differenze tra mean() su Series vs DataFrame

Quando applichi .mean() a una Series, il risultato è un singolo valore scalare che rappresenta la media di quella colonna. Quando la applichi a un DataFrame, il comportamento predefinito è quello di calcolare la media per ciascuna colonna (Series) separatamente, restituendo una nuova Series dove l'indice è il nome della colonna e i valori sono le rispettive medie. È possibile modificare questo comportamento utilizzando il parametro axis.

Parametri utili: axis e skipna

Parametro axis

Il parametro axis è fondamentale quando si lavora con i DataFrame e determina se la media deve essere calcolata lungo le righe o le colonne:

- axis=0 (predefinito per DataFrame): Calcola la media per colonna (lungo le righe).
- axis=1: Calcola la media per riga (lungo le colonne).

```
# Calcoliamo la media per riga (media delle vendite e delle spese per ogni riga)
media_righe = dati_prestazioni.mean(axis=1)
print("\nMedia per ciascuna riga:")
print(media_righe)

# Output previsto:
# Media per ciascuna riga:
# 0    86.666667
# 1   126.666667
# 2   103.333333
# 3    97.500000 (media solo di Vendite_Q2 e Spese_Q1, Vendite_Q1 era NaN)
# 4   115.000000
# dtype: float64
```

Parametro skipna

Come accennato, skipna=True (predefinito) ignora i valori NaN. Se impostato su False, la presenza di un singolo NaN in una Series o in una riga/colonna durante il calcolo della media porterà il risultato a essere NaN.

```
# Calcoliamo la media dell'età includendo i NaN
media_eta_con_nan = eta.mean(skipna=False)
print(f"\nMedia dell'età (con NaN): {media_eta_con_nan}")

# Calcoliamo la media delle colonne del DataFrame includendo i NaN
media_colonne_con_nan = dati_prestazioni.mean(skipna=False)
print("\nMedia per ciascuna colonna (con NaN):")
print(media_colonne_con_nan)

# Output previsto:
# Media dell'età (con NaN): nan
#
# Media per ciascuna colonna (con NaN):
# Vendite_Q1      NaN
# Vendite_Q2    138.0
# Spese_Q1       60.0
# dtype: float64
```

Dataset Pinguini di Palmer

Utilizziamo il dataset dei pinguini di Palmer, reso disponibile da Kristen Gorman e Palmer Station Antarctica LTER. Contiene misurazioni morfologiche di tre specie di pinguini.

```
df = pd.read_csv("data/penguins.csv")
```

Specie

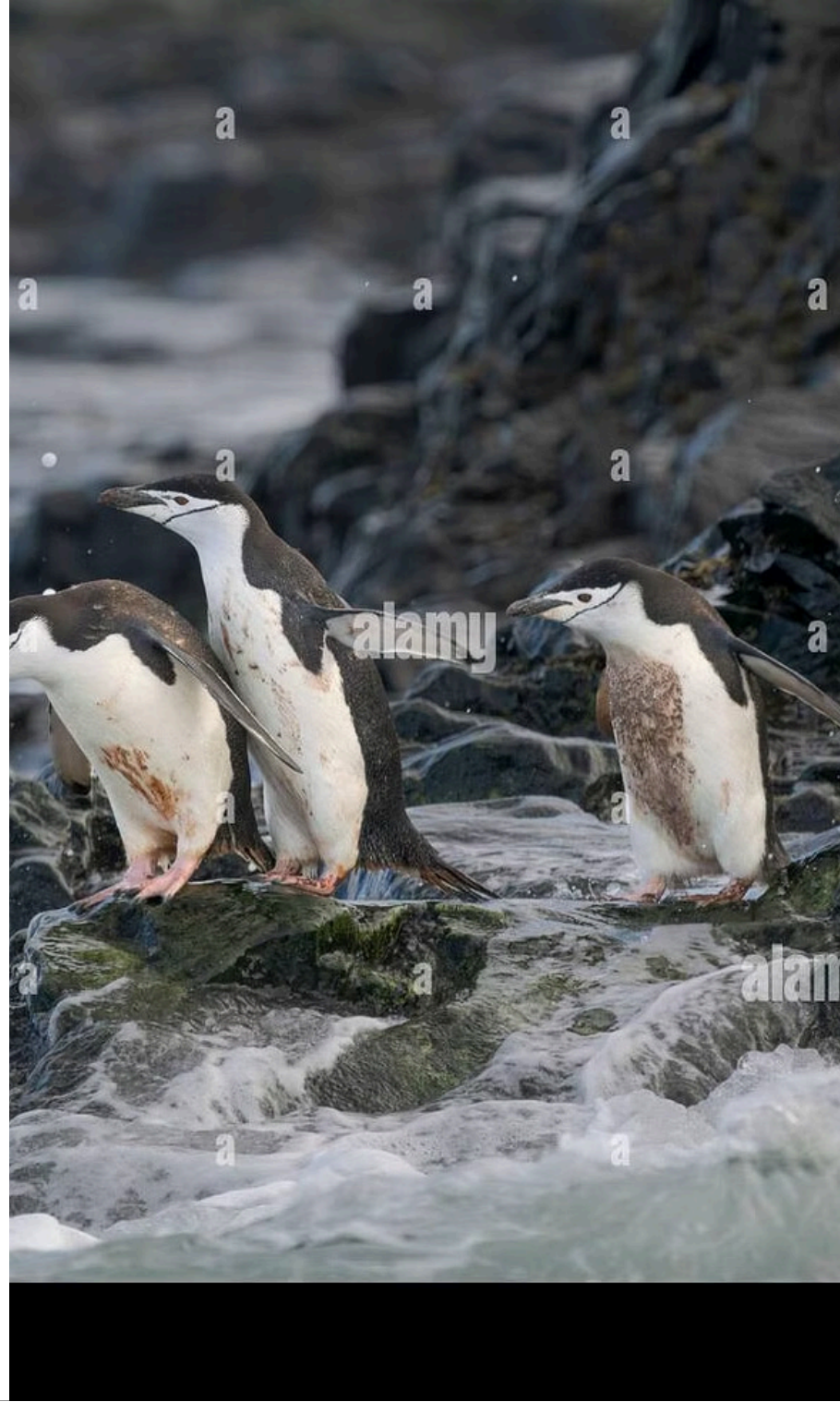
Adélie, Chinstrap e Gentoo

Isole

Biscoe, Dream e Torgersen

Misurazioni

Becco, pinne, massa corporea



Struttura del Dataset

Il dataset contiene 344 osservazioni con 8 variabili:

Variabile	Descrizione
species	Specie di pinguino (Adélie, Chinstrap, Gentoo)
island	Isola nell'arcipelago Palmer (Biscoe, Dream, Torgersen)
bill_length_mm	Lunghezza del becco in millimetri
bill_depth_mm	Profondità del becco in millimetri
flipper_length_mm	Lunghezza delle pinne in millimetri
body_mass_g	Massa corporea in grammi
sex	Sesso (female, male)
year	Anno dello studio

Attributi Fondamentali

I **DataFrame** hanno attributi essenziali per comprendere la struttura dei dati:

.dtypes

Tipo di dati in ogni colonna (object, float64, int64)

.shape

Dimensioni del DataFrame (righe, colonne)

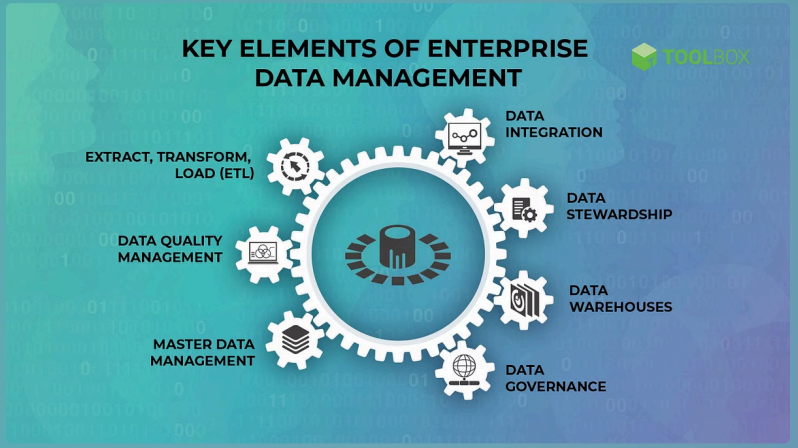
.columns

Nomi delle colonne del DataFrame

.index

Etichette delle righe del DataFrame

```
df.shape # (344, 8)
df.columns # Index(['species', 'island', ...])
```



Metodi di Ispezione

.describe()

Statistiche descrittive per colonne numeriche: conteggio, media, deviazione standard, quartili

.info()

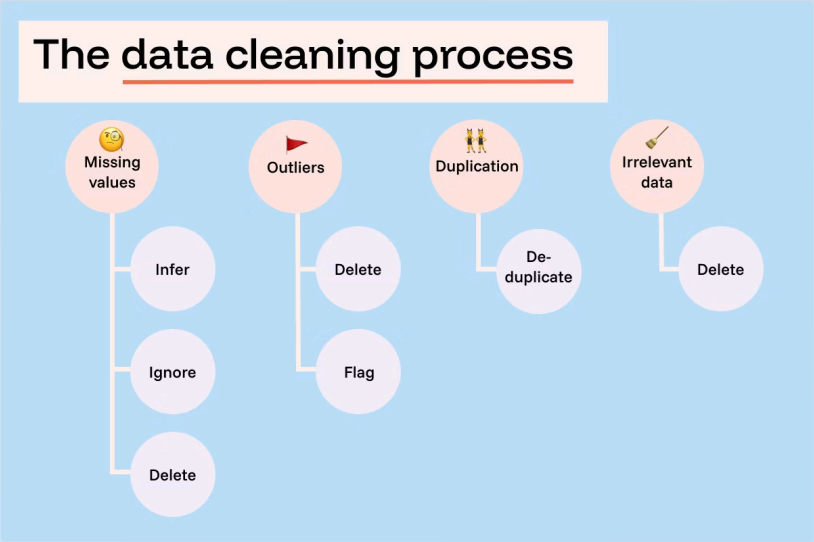
Informazioni generali: tipi di dati, valori non nulli, utilizzo memoria

.head() / .tail()

Visualizza prime/ultime righe per anteprima rapida dei dati

.isnull().sum()

Conta i valori mancanti per ogni colonna



Gestione Dati Mancanti

I dati mancanti sono comuni nei dataset reali. Il dataset pinguini presenta alcuni valori mancanti che devono essere gestiti prima dell'analisi.

```
df.isnull().sum()
# species           0
# island            0
# bill_length_mm    2
# bill_depth_mm     2
# flipper_length_mm 2
# body_mass_g       2
# sex              11
# year              0
```

Il metodo `.dropna(inplace=True)` rimuove tutte le righe contenenti almeno un valore mancante, riducendo il dataset da 344 a 333 osservazioni.

Rinominare le Colonne

È possibile rinominare le colonne usando il metodo `.rename()` con un dizionario che mappa i nomi vecchi a quelli nuovi:

```
df1 = df.copy()
df1.rename(columns={
    "sex": "gender",
    "year": "year_of_the_study"
}, inplace=True)
```

Regole per i nomi delle variabili in Python: Devono iniziare con lettera o underscore, possono contenere solo caratteri alfanumerici e underscore, sono case-sensitive. Gli spazi non sono consentiti.

Selezione di Colonne

Esistono diversi modi per selezionare colonne da un DataFrame:

Singola Colonna

```
df["bill_length_mm"]  # Restituisce una Series
df.bill_length_mm     # Notazione alternativa
```

Multiple Colonne

```
df[["bill_length_mm", "species"]]  # Restituisce DataFrame
```

La selezione di una singola colonna restituisce una Series, mentre la selezione di multiple colonne restituisce un nuovo DataFrame.

Selezione di Righe

Le righe possono essere selezionate usando diversi metodi:

01

Slicing

`df[0:3]` - Prime tre righe

02

iloc

`df.iloc[0]` - Prima riga per posizione

03

loc

`df.loc[4, "body_mass_g"]` - Valore specifico

L'attributo `iloc` usa posizioni numeriche, mentre `loc` usa etichette di righe e colonne.

Filtraggio Condizionale

Il **filtraggio booleano** permette di selezionare **righe** basate su condizioni logiche:

```
only_torgersen = df[df["island"] == "Torgersen"]
```

È possibile combinare **multiple condizioni** usando operatori logici:

```
df.loc[(df["island"] == "Torgersen") & (df["sex"] == "female")]
```

Attenzione: Usare `&` per "e" e `|` per "oppure". Le parentesi sono essenziali per raggruppare le condizioni correttamente.

Metodo .query()

Il metodo `.query()` offre una sintassi più pulita per il filtraggio:

```
eval_string = "island == 'Torgersen' & sex == 'female' & year != 2009"
df.query(eval_string)[["bill_depth_mm", "flipper_length_mm"]]
```

Operatore 'in'

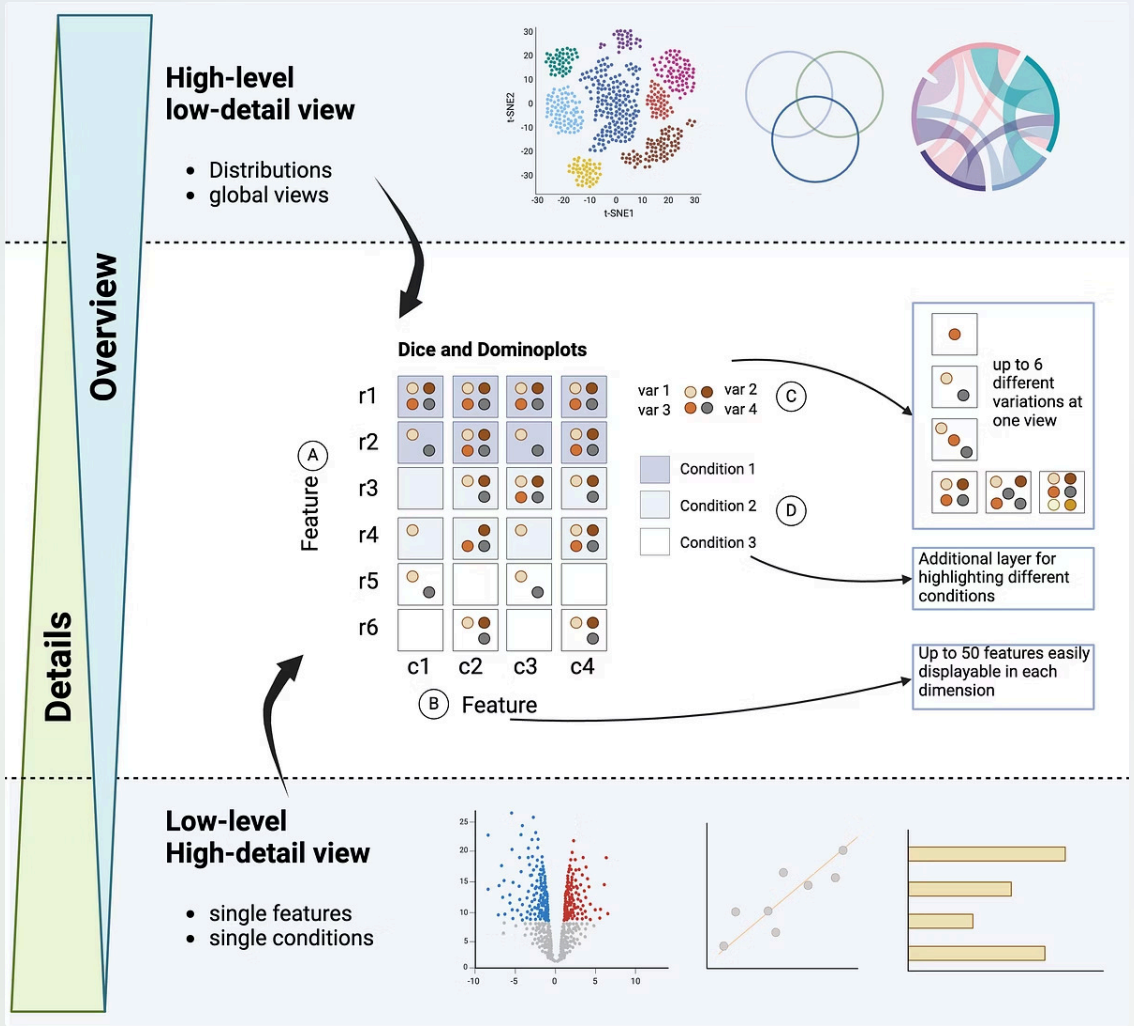
```
df.query("island in ['Torgersen', 'Dream']")
```

Relazioni tra Colonne

```
df.query("bill_length_mm > 3*bill_depth_mm")
```

Variabili Esterne

```
outside_var = 21
df.query("bill_depth_mm > @outside_var")
```



Selezione Casuale

Il metodo `.sample()` permette di ottenere un sottoinsieme casuale di righe:

```
df_sample = df.sample(4)  # 4 righe casuali
```

Senza Rimessa

replace=False (default) - Ogni riga può essere selezionata una sola volta

Con Rimessa

replace=True - Le righe possono essere selezionate multiple volte

Utile per creare campioni rappresentativi per analisi esplorative o testing.

Eliminazione di Colonne

Il metodo `.drop()` rimuove colonne specificate. Le espressioni regolari semplificano la selezione:

```
mask = df.columns.str.contains("mm$|year", regex=True)
columns_to_drop = df.columns[mask]
df_new = df.drop(columns=columns_to_drop)
```

$ (Fine Stringa)

Trova colonne che finiscono con pattern specifico

^ (Inizio Stringa)

Trova colonne che iniziano con pattern specifico

| (Oppure)

Combina multiple condizioni di ricerca

Creazione di Nuove Colonne

Esistono diversi approcci per creare nuove colonne derivate:

1

Metodo .assign()

```
df = df.assign(
    bill_difference=lambda x: x.bill_length_mm - x.bill_depth_mm,
    bill_ratio=lambda x: x.bill_length_mm / (x.body_mass_g / 1000)
)
```

2

Assegnazione Diretta

```
df["bill_ratio2"] = df["bill_length_mm"] / (df["body_mass_g"] / 1000)
```

Il metodo `.assign()` permette di creare multiple colonne simultaneamente usando lambda functions.

Colonne Condizionali

La funzione `np.where()` crea colonne basate su condizioni True/False:

```
np.where(condition, value_if_true, value_if_false)
```

Esempio pratico:

```
df = pd.DataFrame({'A': [-1, 2, 3, -4], 'B': [5, 6, 0, 8]})
df['C'] = df['A'].where(df['A'] > 0, df['B'])
```

Risultato: la colonna C contiene il valore di A quando positivo, altrimenti il valore di B.

Formato Wide vs Long

I dati possono essere strutturati in due formati principali:

Formato Wide

Ogni riga è un'osservazione, ogni variabile ha multiple colonne (es. anni diversi)

Formato Long

Ogni riga è un'osservazione singola, ogni colonna è una variabile

Il formato long è spesso richiesto per analisi statistiche e visualizzazioni avanzate.

Trasformazione con .melt()

La funzione `.melt()` trasforma dati dal formato wide al formato long:

```
scores = {
    "Name": ["Maria", "Carlo", "Giovanna", "Irene"],
    "ID": [1, 2, 3, 4],
    "2017": [85, 87, 89, 91],
    "2018": [96, 98, 100, 102],
    "2019": [100, 102, 106, 106]
}
wide_data = pd.DataFrame(scores)

long_data = wide_data.melt(
    id_vars=["Name", "ID"],
    var_name="Year",
    value_name="Score"
)
```

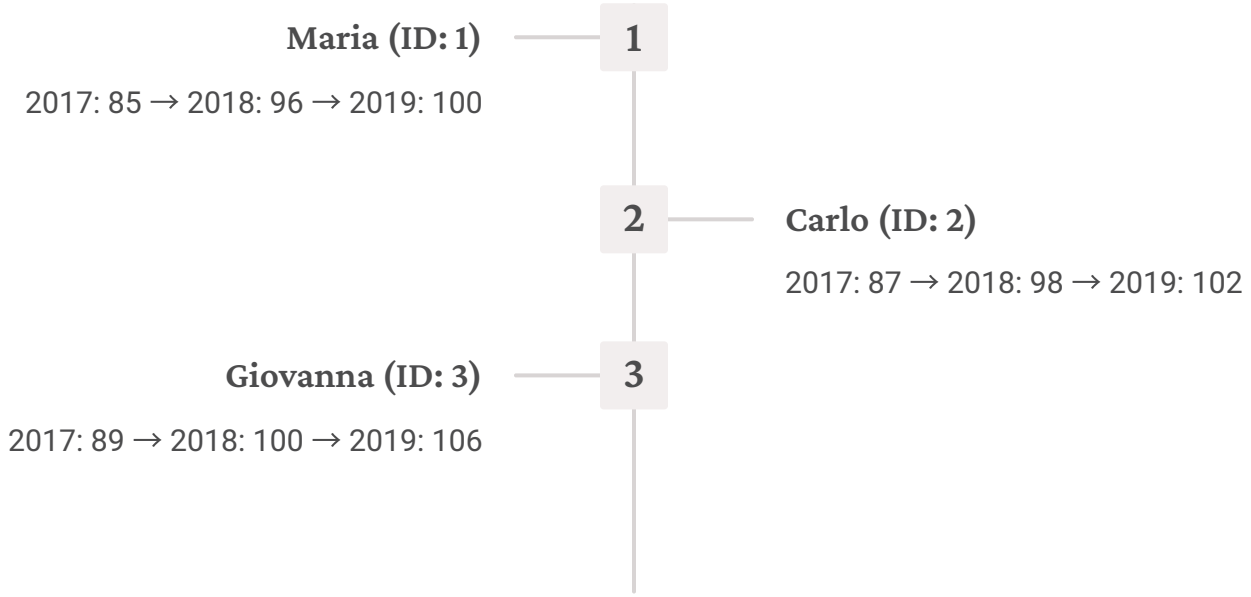
I dati vengono ristrutturati mantenendo le colonne identificative e creando nuove colonne per variabili e valori.

Ordinamento dei Dati

Il metodo `.sort_values()` migliora la leggibilità ordinando i dati secondo criteri specifici:

```
long_data.sort_values(by=["ID", "Year"])
```

Questo ordina prima per ID (partecipante) e poi per Year (anno), creando una sequenza logica che facilita l'interpretazione dei dati longitudinali.



Riepilogo e Prossimi Passi

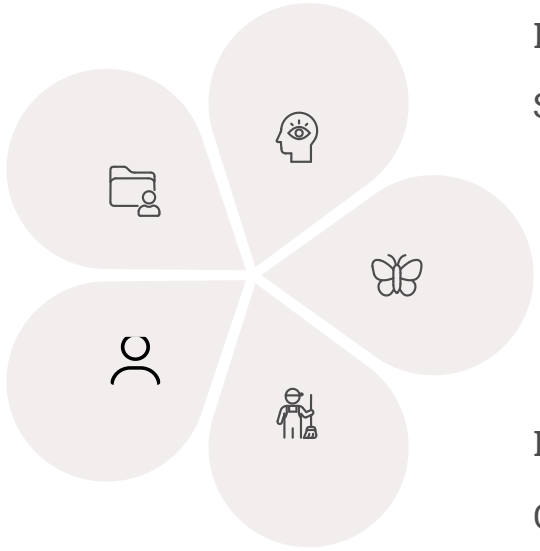
Abbiamo esplorato i concetti fondamentali di Pandas: dalle Series ai DataFrame, dalla lettura dei dati alla manipolazione avanzata. Questi strumenti costituiscono la base per l'analisi dei dati in Python.

Importazione

Lettura da file CSV, Excel, JSON

Ristrutturazione

Formato wide e long



Filtraggio

Selezione condizionale dei dati

Trasformazione

Creazione e modifica colonne

Pulizia

Gestione dati mancanti

Per approfondimenti, consultare il capitolo 10 di "Python for Data Analysis, 3E" di Wes McKinney.