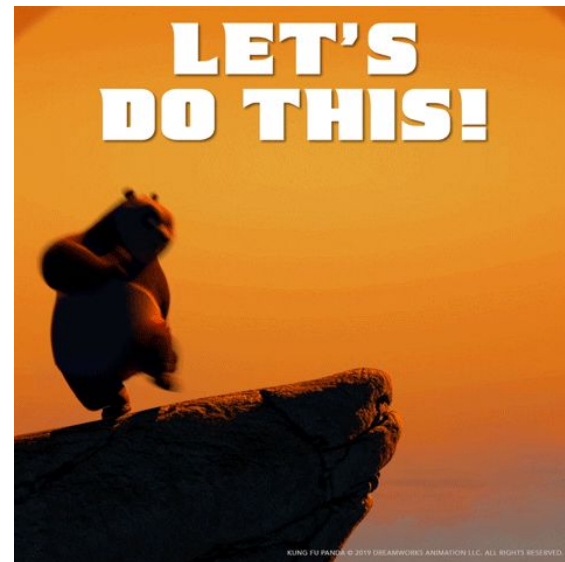


Python

**GIORNO 6 – Pandas – Pulizia dei
dati e gestione dei nulli**

Agenda:

- ❑ Pandas - Pulizia dei dati
- ❑ Pandas - Gestione dei nulli



Perché?

- ❑ I dati puliti sono un'utopia. A causa di problematiche tecniche, strutturali, metodologiche e/o di errori umani è molto raro che i dati che riceviamo siano da subito puliti e completamente utilizzabili.
- ❑ Per poter portare avanti un'analisi significativa diventa necessario sapere come gestire i dati "sporchi" o nulli.

Alla fine di questa lezione saprete:

- ❑ Gestire i dati duplicati
- ❑ Utilizzare i metodi di Pandas per manipolare i dati testuali
- ❑ Gestire la pulizia delle date e dei dati numerici
- ❑ Le tipologie principali di dati nulli in Pandas
- ❑ Come gestire i valori nulli nelle situazioni più comuni

Data Cleaning

Il data cleaning è il processo di identificazione, correzione e eliminazione di errori, inconsistenze o dati incompleti all'interno di un set di dati al fine di migliorarne la qualità e l'affidabilità per l'analisi e l'elaborazione successiva.



Data Cleaning

I problemi che possiamo riscontrare in un set di dati sono vari e così anche i metodi per gestirli: per convenienza li raggruppiamo in base alla tipologia di dato a cui fanno riferimento e andiamo a vedere nel dettaglio come gestire:

- ❑ Righe duplicate
- ❑ Dati testuali
- ❑ Dati temporali
- ❑ Dati numerici
- ❑ Dati nulli

Righe duplicate

Può capitare di incontrare righe duplicate nel nostro set di dati. A volte può persino succedere che, in base alle operazioni che effettuiamo per processare i dati, siamo proprio noi a duplicare delle righe, o dei subset di esse.

Andiamo a creare un semplice DataFrame di esempio:

```
df = pd.DataFrame({"Studente": ["Andrea", "Maria", "Marco", "Silvia", "Andrea"],  
                  "Corso": ["WebDev", "WebDev", "DataAnalysis", "CyberSecurity", "WebDev"]})  
df
```

✓ 0.0s

	Studente	Corso
0	Andrea	WebDev
1	Maria	WebDev
2	Marco	DataAnalysis
3	Silvia	CyberSecurity
4	Andrea	WebDev

Ad indice 0 ed indice 4 abbiamo una riga duplicata, cioè una riga che presenta gli stessi valori nelle stesse colonne.

Righe duplicate

Con un numero di righe così piccolo è facile trovare i duplicati anche ad occhio, ma come facciamo quando il numero di dati aumenta?

Possiamo utilizzare il metodo `.duplicated()` di Pandas:

```
df.duplicated()
✓ 0.0s
0    False
1    False
2    False
3    False
4     True
dtype: bool
```

Il metodo `.duplicated()` restituisce una Series in cui, per ogni indice del `DataFrame`, troviamo un valore booleano ad indicarci se la riga è già presente nel `DataFrame` stesso. Possiamo quindi usarla anche per filtrare:

```
df.loc[df.duplicated()]
✓ 0.0s
```

	Studente	Corso
4	Andrea	WebDev

Righe duplicate

Il metodo `.duplicated()` accetta due parametri:

- ❑ `subset=`: la colonna, o lista di colonne, su cui effettuare il controllo. Default: `None`
- ❑ `keep=`: quale delle righe segnare come duplicata. Può essere settato come `"first"`, `"last"` o `False`. Default: `"first"`

Righe duplicate

Per trovare i duplicati soltanto della colonna "Corso" utilizzeremo il parametro `subset=`:

```
df.loc[df.duplicated(subset="Corso")]
```

✓ 0.0s

	Studente	Corso
1	Maria	WebDev
4	Andrea	WebDev

In questo modo **tutti a parte la prima occorrenza** di ogni valore verranno segnati come duplicati. Per marcare come duplicata anche la riga "originale" settiamo il parametro `keep=False`:

```
df.loc[df.duplicated(subset="Corso", keep=False)]
```

✓ 0.0s

	Studente	Corso
0	Andrea	WebDev
1	Maria	WebDev
4	Andrea	WebDev

Se invece vogliamo che venga interpretata come riga "originale" l'ultima, settiamo `keep="last"`

```
df.loc[df.duplicated(subset="Corso", keep="last")]
```

✓ 0.0s

	Studente	Corso
0	Andrea	WebDev
1	Maria	WebDev

Righe duplicate

Solitamente i duplicati si eliminano: Pandas ci viene in aiuto con il metodo `.drop_duplicates()`:

```
df
✓ 0.0s

  Studente  Corso
0  Andrea  WebDev
1  Maria   WebDev
2  Marco   DataAnalysis
3  Silvia  CyberSecurity
4  Andrea  WebDev

df.drop_duplicates()
✓ 0.0s

  Studente  Corso
0  Andrea  WebDev
1  Maria   WebDev
2  Marco   DataAnalysis
3  Silvia  CyberSecurity

df.drop_duplicates(subset="Corso")
✓ 0.0s

  Studente  Corso
0  Andrea  WebDev
2  Marco   DataAnalysis
3  Silvia  CyberSecurity

df.drop_duplicates(subset="Corso", keep=False)
✓ 0.0s

  Studente  Corso
2  Marco   DataAnalysis
3  Silvia  CyberSecurity

df.drop_duplicates(subset="Corso", keep="last")
✓ 0.0s

  Studente  Corso
2  Marco   DataAnalysis
3  Silvia  CyberSecurity
4  Andrea  WebDev
```

Righe duplicate

Come abbiamo visto `.drop_duplicates()` accetta gli stessi parametri di `.duplicated()`

Il metodo però **non va ad aggiornare il DataFrame originale**: se dopo aver fatto `.drop_duplicates()` andiamo a guardare ancora il nostro DataFrame, i dati sono ancora tutti lì:

```
df.drop_duplicates()
✓ 0.0s
```

	Studiante	Corso
0	Andrea	WebDev
1	Maria	WebDev
2	Marco	DataAnalysis
3	Silvia	CyberSecurity

```
df
✓ 0.0s
```

	Studiante	Corso
0	Andrea	WebDev
1	Maria	WebDev
2	Marco	DataAnalysis
3	Silvia	CyberSecurity
4	Andrea	WebDev

Righe duplicate

Esistono due metodi per modificare il nostro DataFrame originale: il primo è quello di aggiungere come parametro `inplace=True`

```
df
✓ 0.0s
```

	Studiante	Corso
0	Andrea	WebDev
1	Maria	WebDev
2	Marco	DataAnalysis
3	Silvia	CyberSecurity
4	Andrea	WebDev

```
df.drop_duplicates(inplace=True)
✓ 0.0s
```

```
df
✓ 0.0s
```

	Studiante	Corso
0	Andrea	WebDev
1	Maria	WebDev
2	Marco	DataAnalysis
3	Silvia	CyberSecurity

Questo metodo è **fortemente sconsigliato** per delle ragioni tecniche relative a come Python e Pandas gestiscono gli oggetti in memoria, al punto che Pandas ha in progetto di rimuoverlo gradualmente nelle prossime versioni.

Righe duplicate

Perché presentarvi un metodo sconsigliato? Perché se avrete a che fare con una codebase magari un po' datata, o scritta senza una profonda attenzione all'ottimizzazione, potrete incontrare questo parametro.

Come fare allora?

La best practice è: **riassegnare alla stessa variabile il risultato del metodo `.drop_duplicates()`**

```
df
✓ 0.0s
```

	Studiante	Corso
0	Andrea	WebDev
1	Maria	WebDev
2	Marco	DataAnalysis
3	Silvia	CyberSecurity
4	Andrea	WebDev

```
df = df.drop_duplicates()
✓ 0.0s
```

```
df
✓ 0.0s
```

	Studiante	Corso
0	Andrea	WebDev
1	Maria	WebDev
2	Marco	DataAnalysis
3	Silvia	CyberSecurity

Best Practice

Questa pratica ci accompagnerà anche per tutti i prossimi metodi che modificano in qualche modo il `DataFrame`; questi **non rendono permanente la modifica**, ma restituiscono in output una versione diversa del `DataFrame`, oppure una `Series` se sono applicati ad una sola colonna o in generale se il risultato è monodimensionale.

Per memorizzare la nuova versione in output possiamo semplicemente assegnarla ad una nuova variabile, oppure se vogliamo rendere permanente la modifica possiamo sovrascrivere la variabile esistente.

Ricordiamoci sempre che **la best practice è riassegnare alla stessa variabile, o ad una nuova, il risultato del metodo**.

Se il metodo viene applicato ad una singola colonna, si può assegnare effettuando:

```
pandas.DataFrame["colonna"] = pandas.DataFrame["colonna"].metodo()
```

Dati testuali

Per andare ad agire sui dati testuali (cioè le stringhe), abbiamo una serie di metodi tra cui:

- ❑ `.str.upper()`, `.str.lower()`: Trasforma i caratteri in maiuscolo o minuscolo.
- ❑ `.str.strip()`, `.str.lstrip()`, `.str.rstrip()`: Rimuove gli spazi vuoti dall'inizio o dalla fine della stringa.
- ❑ `.str.split()`: Suddivide una stringa in base a un delimitatore specificato.
- ❑ `.str.contains()`: Restituisce una maschera booleana indicante se ogni stringa contiene un determinato pattern.
- ❑ `.str.replace()`: Sostituisce parte di una stringa con un'altra.
- ❑ `.str.extract()`: Estrae sottostringhe che corrispondono a un pattern specificato.

Dati testuali

Per fare degli esempi utilizzeremo il database "Titanic" che si trova sul server SQL della scuola, e che contiene una sola tabella, chiamata `titanic`.

Per accedervi dobbiamo modificare il nome del database nel file `.env`, modificando la riga:

```
dbname="Titanic"
```

oppure aggiungendone una nuova:

```
dbname2="Titanic"
```

Naturalmente in questo secondo caso nel nostro codice Python dovremo modificare la riga:

```
dbname = os.getenv("dbname2")
```

Una buona pratica, tuttavia, è quella di creare **cartelle diverse per ogni progetto** su cui si sta lavorando, e inserire un `.env` diverso in ognuno di essi che contiene i dati relativi **solo** al database specifico per quel progetto.

Questo permette di avere un ambiente di lavoro più ordinato che evita confusione tra i progetti.

.str.upper(), .str.lower()

Modifichiamo il case (maiuscolo o minuscolo) dei nomi dei passeggeri del Titanic:

```
query = """SELECT * FROM titanic"""
df = pd.read_sql(query, db_engine)
df.head()
```

✓ 0.5s

	PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.25		S	0	10-4-1912
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.28	C85	C	1	11-4-1912
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.93		S	1	10-4-1912
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.10	C123	S	1	10-4-1912
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.05		S	0	10-4-1912

```
df["Name"] = df["Name"].str.upper()
df.head()
```

✓ 0.0s

	PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate
0	1	3	BRAUND, MR. OWEN HARRIS	male	22.0	1	0	A/5 21171	7.25		S	0	10-4-1912
1	2	1	CUMINGS, MRS. JOHN BRADLEY (FLORENCE BRIGGS TH...	female	38.0	1	0	PC 17599	71.28	C85	C	1	11-4-1912
2	3	3	HEIKKINEN, MISS. LAINA	female	26.0	0	0	STON/O2. 3101282	7.93		S	1	10-4-1912
3	4	1	FUTRELLE, MRS. JACQUES HEATH (LILY MAY PEEL)	female	35.0	1	0	113803	53.10	C123	S	1	10-4-1912
4	5	3	ALLEN, MR. WILLIAM HENRY	male	35.0	0	0	373450	8.05		S	0	10-4-1912

.str.split()

Suddividiamo il nome di ogni passeggero in più elementi, specificando come pattern di separazione il carattere " , " (se non specifichiamo alcun pattern, il default è uno whitespace, " ") e creiamo una nuova colonna per la lista che ne risulta

```
query = """SELECT * FROM titanic"""
df = pd.read_sql(query, db_engine)
df.head()
```

✓ 0.2s

	PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.25		S	0	10-4-1912
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.28	C85	C	1	11-4-1912
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.93		S	1	10-4-1912
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.10	C123	S	1	10-4-1912
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.05		S	0	10-4-1912

```
df["NameSplitted"] = df["Name"].str.split(pat=",")
df.head()
```

✓ 0.0s

	PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate	NameSplitted
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.25		S	0	10-4-1912	[Braund, Mr. Owen Harris]
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.28	C85	C	1	11-4-1912	[Cumings, Mrs. John Bradley (Florence Briggs ...
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.93		S	1	10-4-1912	[Heikkinen, Miss. Laina]
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.10	C123	S	1	10-4-1912	[Futrelle, Mrs. Jacques Heath (Lily May Peel)]
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.05		S	0	10-4-1912	[Allen, Mr. William Henry]

Docs: <https://pandas.pydata.org/docs/reference/api/pandas.Series.str.split.html>

.str.contains()

il metodo `.str.contains()` restituisce una `Series` booleana, come il metodo `.duplicated()`

Proviamo ad usarlo per filtrare i nomi dei passeggeri che contengono la sottostringa "Mr. "

Il parametro `regex=` è settato `True` di default. Se vogliamo che il nostro pattern voglia interpretato letteralmente e non come una regular expression, dobbiamo specificare `regex=False`

```
query = """SELECT * FROM titanic"""
df = pd.read_sql(query, db_engine)
df.head()
```

✓ 0.2s

PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate	
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.25		S	0	10-4-1912
1	2	1	Cummings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.28	C85	C	1	11-4-1912
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/OZ. 3101282	7.93		S	1	10-4-1912
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.10	C123	S	1	10-4-1912
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.05		S	0	10-4-1912

```
df.loc[df["Name"].str.contains("Mr.", regex=False)]
```

✓ 0.0s

PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate	
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.25		S	0	10-4-1912
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.05		S	0	10-4-1912
5	6	3	Moran, Mr. James	male	0.0	0	0	330877	8.46		Q	0	11-4-1912
6	7	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.86	E46	S	0	10-4-1912
12	13	3	Saunderscock, Mr. William Henry	male	20.0	0	0	A/5. 2151	8.05		S	0	10-4-1912
...
1297	1298	2	Ware, Mr. William Jeffery	male	23.0	1	0	28666	10.50		S	0	10-4-1912
1298	1299	1	Widener, Mr. George Dunton	male	50.0	1	1	113503	211.50	C80	C	0	11-4-1912
1304	1305	3	Spector, Mr. Woolf	male	0.0	0	0	A.S. 3236	8.05		S	0	10-4-1912
1306	1307	3	Saether, Mr. Simon Sivertsen	male	38.5	0	0	SOTON/O.Q. 3101262	7.25		S	0	10-4-1912
1307	1308	3	Ware, Mr. Frederick	male	0.0	0	0	359309	8.05		S	0	10-4-1912

Docs: <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.str.contains.html>

.str.replace()

Permette di sostituire una substringa con un valore.

Sostituiamo le occorrenze di "Mr ." nei nomi dei passeggeri con "Signor":

```
query = """SELECT * FROM titanic"""
df = pd.read_sql(query, db_engine)
df.head()
```

✓ 0.2s

PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate	
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.25		S	0	10-4-1912
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.28	C85	C	1	11-4-1912
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.93		S	1	10-4-1912
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.10	C123	S	1	10-4-1912
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.05		S	0	10-4-1912

```
df["Name"] = df["Name"].str.replace("Mr.", "Signor")
df
```

✓ 0.0s

PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate	
0	1	3	Braund, Signor Owen Harris	male	22.0	1	0	A/5 21171	7.25		S	0	10-4-1912
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.28	C85	C	1	11-4-1912
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.93		S	1	10-4-1912
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.10	C123	S	1	10-4-1912
4	5	3	Allen, Signor William Henry	male	35.0	0	0	373450	8.05		S	0	10-4-1912

.str.extract()

Estrae una sottostringa che corrisponde ad un pattern, che deve necessariamente essere una **regular expression**.

Restituisce un `DataFrame`; per avere una `Series` specifichiamo `expand=False`:

```
query = """SELECT * FROM titanic"""
df = pd.read_sql(query, db_engine)
df.head()
```

✓ 0.2s

	PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.25		S	0	10-4-1912
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.28	C85	C	1	11-4-1912
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.93		S	1	10-4-1912
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.10	C123	S	1	10-4-1912
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.05		S	0	10-4-1912

```
df["Name"] = df["Name"].str.extract(r'(?=\s)', expand=False)
df
```

✓ 0.0s

	PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate
0	1	3	Braund	male	22.0	1	0	A/5 21171	7.25		S	0	10-4-1912
1	2	1	Cumings	female	38.0	1	0	PC 17599	71.28	C85	C	1	11-4-1912
2	3	3	Heikkinen	female	26.0	0	0	STON/O2. 3101282	7.93		S	1	10-4-1912
3	4	1	Futrelle	female	35.0	1	0	113803	53.10	C123	S	1	10-4-1912
4	5	3	Allen	male	35.0	0	0	373450	8.05		S	0	10-4-1912

Docs: <https://pandas.pydata.org/docs/reference/api/pandas.Series.str.extract.html>

Dati testuali - RegEx

Prima di vedere i metodi specifici di Pandas per manipolare i dati testuali, è necessario che introduciamo il concetto di **espressione regolare**

Quando ci troviamo a dover fare delle operazioni sulle stringhe di testo, le Regular Expression (in breve RegEx) permettono operazioni di ricerca e sostituzione e manipolazione dettagliate e approfondite.

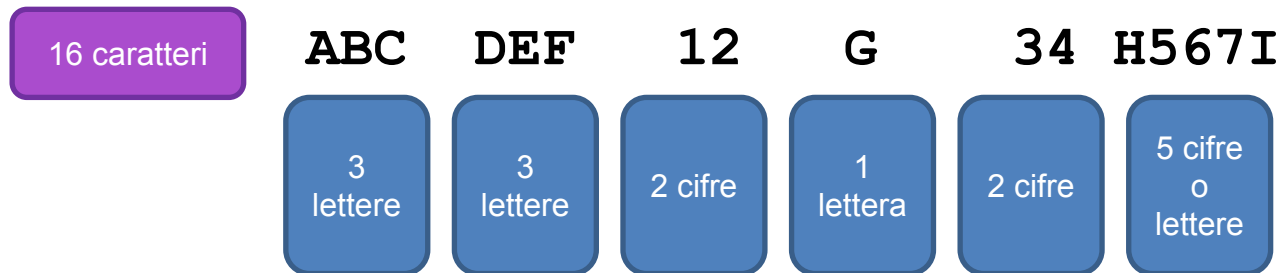


Dati testuali - RegEx

Se ci viene presentato un testo qualsiasi, e ci viene richiesto di leggerlo e di identificare tutti i numeri di telefono al suo interno, oppure i codici fiscali, oppure tutti gli indirizzi email, siamo perfettamente in grado di farlo.

Come facciamo, visto che ovviamente contengono numeri o lettere diversi (telefono o CF), o addirittura possono essere di lunghezza diversa (email)?

Possiamo farlo perché hanno un **pattern specifico** che si può identificare, ad esempio per il CF:



Dati testuali - RegEx

Per "spiegare" ad una macchina questi pattern possiamo usare le **Regular Expression** (RegEx o RegExp), che si presentano come **stringhe di testo composte da caratteri e metacaratteri**, dove i caratteri rappresentano sé stessi, mentre i metacaratteri sono caratteri che hanno un significato proprio e rappresentano classi di caratteri o comportamenti speciali.

La combinazione degli uni e degli altri ci permette di definire dei **pattern** da identificare all'interno di una stringa di testo.

Dati testuali - RegEx

Esempi di metacaratteri e sintassi RegEx:

- `.` un carattere qualsiasi
- `\d` un carattere alfanumerico (anche underscore)
- `\w` una cifra
- `\s` un carattere spazio (anche tab o newline)
- `+` il pattern precedente, ripetuto una o più volte
- `*` il pattern precedente, ripetuto zero o più volte
- `?` il pattern precedente può esserci oppure no
- `{n, m}` il pattern precedente ripetuto da n a m volte
- `[]` contengono un insieme di caratteri di interesse
- `^` inizio della riga
- `$` fine della riga
- `-` range di caratteri (da usare nelle parentesi quadre)

Dati testuali - RegEx

Regex: `ciao`

Riconosce il pattern:

✓ ciao

✗ miao

`[cm]iao`

✓ ciao

✓ miao

✗ iao

✗ kiao

`u.a`

✓ usa

✓ una

✓ u0a

✓ u#a

✓ usato (contiene il pattern)

✗ uffa (ci sono più di un carattere tra "u" e "a")

✗ ua (non ci sono caratteri tra "u" e "a")

Dati testuali - RegEx

`fatt?o`

✓ fatto

✓ fato

✗ favo

✗ fattto

`\d{3}`

✓ Blink 182

✓ La carica dei 101

✓ Nel 2005 è successo questo (esiste una sequenza di tre cifre)

✗ Ci sono 12 mesi

✗ La temperatura è di 9 gradi

`^[A-Z]`

✓ Ciao a tutti (c'è una lettera maiuscola ad inizio riga)

✓ Buongiorno (c'è una lettera maiuscola ad inizio riga)

✗ 1 Lettera (c'è una cifra ad inizio riga)

✗ salve (non c'è una lettera maiuscola ad inizio riga)

Dati testuali - RegEx

`\d+`

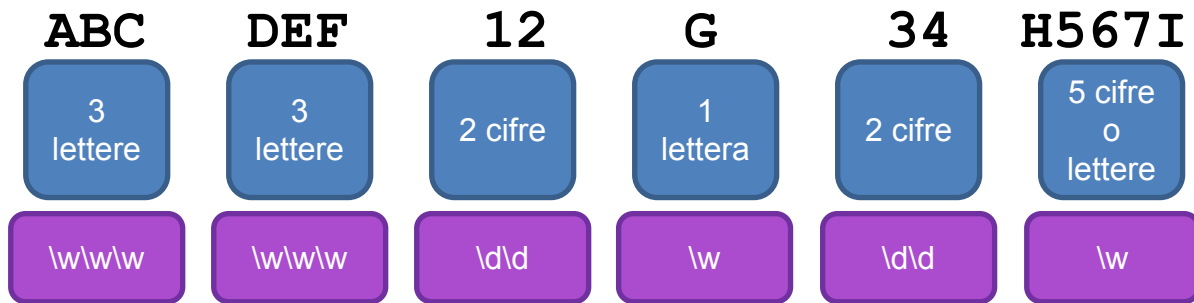
- ✓ Compra 3 pacchi di patatine
- ✓ Sono le 20:30
- ✓ Il budget è di \$1000.00
- ✗ Compra tre pacchi di patatine
- ✗ Sono le venti e trenta

`\w+@\w+\.[a-z]{2,3}`

- `\w+` uno o più caratteri alfanumerici
 - `@` una chiocciola
 - `\.` un punto (letterale)
 - `[a-z]{2,3}` due o tre lettere minuscole
- ✓ `nomecognome@provider.com`
 - ✓ `pinco_pallino@epicode.it`
 - ✓ `user1234@sitoweb2.gov`
 - ✗ `utente@@gmail.com` (non c'è una sola chiocciola)
 - ✗ `nome@sitoweb.comm` (l'estensione del provider ha 4 caratteri)

Dati testuali - RegEx

Esempio per il CF:



Oppure:



Poiché `\w` "cattura" anche le cifre; conoscendo i dettagli delle componenti di un CF, un modo ancora più preciso sarebbe:



Date

Per agire sulle date ci sono varie funzioni e metodi:

- ❑ `pd.to_datetime()`: converte una stringa di testo in un formato data
- ❑ `.dt.year`, `.dt.month`, `.dt.day`: estrarre una parte della data
- ❑ `.dt.dayofweek`, `.dt.dayofyear`: contestualizza la data

pd.to_datetime()

`pd.to_datetime()` è una funzione fondamentale. Molto spesso le date sono inserite o interpretate come stringhe, impedendoci di effettuare tutte le operazioni possibili/necessarie su questo speciale tipo di dato.

Usando `pd.to_datetime()` possiamo convertire le stringhe in formati data utilizzando vari parametri, tra cui i più importanti sono `dayfirst=` e `yearfirst=` che specificano come la stringa debba essere interpretata, in particolare in situazioni quali notazione anglosassone (mese/giorno/anno) o in situazioni ambigue (ad esempio 11/09/10).

Per tutti i parametri possibili, invece, riferiamoci alla documentazione di Pandas:

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

`pd.to_datetime()`

Utilizziamo `pd.to_datetime()` per convertire la data di imbarco dei passeggeri da stringa a formato data; poiché le date sono ambigue (tutti i giorni sono <12) specifichiamo `dayfirst=True`:

```
query = "SELECT * FROM titanic"

df = pd.read_sql(query, db_engine)

db_engine.close()

df.loc[:, "EmbarkDate"] = pd.to_datetime(df.loc[:, "EmbarkDate"], dayfirst=True)
```

pd.to_datetime()

Nella visualizzazione non sembra cambiato molto, se non che ora l'anno e il mese vengono prima del giorno, ma se andiamo ad indagare la tipologia di dato tramite l'attributo `.dtype` vediamo che è cambiato, e **passiamo da un `dtype('O')`**, che sta per Object, che è come Pandas interpreta le stringhe o i dati misti, **a un `dtype('<M8[ns]')`**, che è un datatype specifico di NumPy per la gestione delle date.

.dt.year, .dt.month, .dt.day, .dt.dayofweek, .dt.dayofyear

Per estrarre delle parti specifiche della data, accediamo agli **attributi del date object**.

Estraiamoli dalla data di imbarco e assegniamoli a nuove colonne:

```
query = "SELECT * FROM titanic"
df = pd.read_sql(query, db_engine)
db_engine.close()

df.loc[:, "EmbarkDate"] = pd.to_datetime(df.loc[:, "EmbarkDate"], dayfirst=True)

df.loc[:, "Year"] = df.loc[:, "EmbarkDate"].dt.year

df.loc[:, "Month"] = df.loc[:, "EmbarkDate"].dt.month

df.loc[:, "Day"] = df.loc[:, "EmbarkDate"].dt.day

df.loc[:, "WeekDay"] = df.loc[:, "EmbarkDate"].dt.dayofweek # 0: lunedì, 6: domenica

df.loc[:, "DayOfYear"] = df.loc[:, "EmbarkDate"].dt.dayofyear
```

Docs: <https://pandas.pydata.org/pandas-docs/stable/reference/series.html#datetimelike-properties>

Dati numerici

I principali metodi per pulire dati numerici sono:

- ❑ `.round()`: arrotonda i decimali di un float
- ❑ `.clip()`: stabilisce un limite massimo o minimo per i numeri

Sembrano pochi, ma questo è perché ai dati numerici si possono applicare tutte quelle funzioni base di Python e di NumPy che si occupano di gestire numeri e calcoli, quindi ci sono meno implementazioni specifiche legate a Pandas.

.round()

Usando `.round()` possiamo arrotondare un float, definendo quanti numeri dopo la virgola tenere. Ad esempio, arrotondiamo il costo del biglietto (contenuto nella colonna "Fare") ad un solo numero decimale:

```
query = """SELECT * FROM titanic"""
df = pd.read_sql(query, db_engine)
df.head()
```

✓ 0.2s

PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate	
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.25		S	0	10-4-1912
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.28	C85	C	1	11-4-1912
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.93		S	1	10-4-1912
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.10	C123	S	1	10-4-1912
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.05		S	0	10-4-1912

```
df["Fare"] = df["Fare"].round(1)
df.head()
```

✓ 0.0s

PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate	
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2		S	0	10-4-1912
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.3	C85	C	1	11-4-1912
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9		S	1	10-4-1912
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1	C123	S	1	10-4-1912
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0		S	0	10-4-1912

.clip()

Con `.clip()` possiamo delimitare il range dei nostri numeri, specificando un limite massimo e minimo tramite i parametri `upper=` e `lower=`.

Settiamo come limite massimo 55, e come limite minimo 8:

```
query = """SELECT * FROM titanic"""
df = pd.read_sql(query, db_engine)
df.head()
```

✓ 0.2s

	PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.25		S	0	10-4-1912
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.28	C85	C	1	11-4-1912
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.93		S	1	10-4-1912
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.10	C123	S	1	10-4-1912
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.05		S	0	10-4-1912

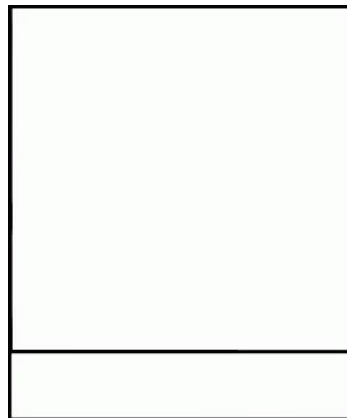
```
df["FareClipped"] = df["Fare"].clip(lower=8, upper=55)
df.head()
```

✓ 0.0s

	PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate	FareClipped
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.25		S	0	10-4-1912	8.00
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.28	C85	C	1	11-4-1912	55.00
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.93		S	1	10-4-1912	8.00
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.10	C123	S	1	10-4-1912	53.10
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.05		S	0	10-4-1912	8.05

Dati nulli

Un dato nullo è un'osservazione mancante o non disponibile, che può influenzare l'accuratezza e l'affidabilità delle analisi statistiche e dei modelli predittivi



Dati nulli

In Pandas i dati nulli si possono presentare in modo **esplicito** e in modo **implicito**.

Quando i dati nulli sono espliciti, le celle sono riempite da valori **NaN** (Not a Number) o **NaT** (Not a Time), che sono datatype specifici di NumPy e Pandas, oppure attraverso dei **None** (il tipo di dato vuoto nativo di Python).

Quando i dati sono in questo formato, Pandas implementa tutta una serie di metodi per gestirli.

Quando i dati nulli sono invece **impliciti** abbiamo prima bisogno di convertirli in un valore che Pandas sappia interpretare come nullo. È esattamente il nostro caso, dove i dati nulli sono rappresentati da stringhe vuote.

Dati nulli

Vediamo infatti che, nonostante abbiamo delle celle vuote, il conteggio dei valori non nulli è uguale al numero delle righe del nostro DataFrame

```
df.head()
```

	PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.25		S	0	10-4-1912
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.28	C85	C	1	11-4-1912
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.93		S	1	10-4-1912
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.10	C123	S	1	10-4-1912
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.05		S	0	10-4-1912


```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1309 entries, 0 to 1308
Data columns (total 13 columns):
#   Column      Non-Null Count  Dtype
---  -
0   PassengerId  1309 non-null   int64
1   PClass       1309 non-null   int64
2   Name         1309 non-null   object
3   Sex          1309 non-null   object
4   Age         1309 non-null   float64
5   SibSp        1309 non-null   int64
6   Parch        1309 non-null   int64
7   Ticket       1309 non-null   object
8   Fare         1309 non-null   float64
9   Cabin        1309 non-null   object
10  Embarked     1309 non-null   object
11  Survived     1309 non-null   int64
12  EmbarkDate   1309 non-null   object
dtypes: float64(2), int64(5), object(6)
memory usage: 133.1+ KB
```

Dati nulli

Andiamo a dare a questi dati nulli il formato tramite cui è più facile gestirli aiutandoci con `.replace()` e NumPy, e vediamo l'impatto su Non-Null Count:

```
import numpy as np

df = df.replace("", None)
df.info()
```

✓ 0.0s

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1309 entries, 0 to 1308
Data columns (total 13 columns):
#   Column      Non-Null Count  Dtype
---  ---
0   PassengerId 1309 non-null   int64
1   PClass      1309 non-null   int64
2   Name        1309 non-null   object
3   Sex         1309 non-null   object
4   Age         1309 non-null   float64
5   SibSp       1309 non-null   int64
6   Parch       1309 non-null   int64
7   Ticket      1309 non-null   object
8   Fare        1309 non-null   float64
9   Cabin       295 non-null    object
10  Embarked     1307 non-null   object
11  Survived     1309 non-null   int64
12  EmbarkDate   1307 non-null   object
dtypes: float64(2), int64(5), object(6)
memory usage: 133.1+ KB
```

Dati nulli - .isna()

Un metodo per indagare i nulli, molto simile a quello dei duplicati, è `.isna()`, che restituisce una maschera, ovvero un `DataFrame` di booleani:

```
df.isna()
```

✓ 0.0s

	PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate
0	False	False	False	False	False	False	False	False	False	True	False	False	False
1	False	False	False	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	True	False	False	False
3	False	False	False	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	True	False	False	False
...

Possiamo applicarlo anche ad una colonna, e in quel caso possiamo usare la maschera per filtrare il `DataFrame`

```
df.loc[df["Embarked"].isna()]
```

✓ 0.0s

	PassengerId	PClass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived	EmbarkDate
61	62	1	Icard, Miss. Amelie	female	38.0	0	0	113572	80.0	B28	NaN	1	NaN
829	830	1	Stone, Mrs. George Nelson (Martha Evelyn)	female	62.0	0	0	113572	80.0	B28	NaN	1	NaN

Dati nulli - .dropna()

Quando il numero dei dati nullo è molto basso rispetto alla totalità della base dati, possiamo semplicemente eliminare le righe. Ad esempio, abbiamo soltanto 2 righe su 1309 con dei valori nulli nella colonna "Embarked", che rappresentano lo 0.15% del totale.

Pandas fornisce il metodo `.dropna()`, a cui possiamo specificare un subset di colonne.

```
df = df.dropna(subset="Embarked")
df.info()
✓ 0.0s

<class 'pandas.core.frame.DataFrame'>
Index: 1307 entries, 0 to 1308
Data columns (total 13 columns):
#   Column      Non-Null Count  Dtype
---  -
0   PassengerId  1307 non-null   int64
1   PClass      1307 non-null   int64
2   Name        1307 non-null   object
3   Sex         1307 non-null   object
4   Age         1307 non-null   float64
5   SibSp       1307 non-null   int64
6   Parch       1307 non-null   int64
7   Ticket      1307 non-null   object
8   Fare        1307 non-null   float64
9   Cabin       293 non-null    object
10  Embarked     1307 non-null   object
11  Survived     1307 non-null   int64
12  EmbarkDate   1307 non-null   object
dtypes: float64(2), int64(5), object(6)
memory usage: 143.0+ KB
```

Dati nulli

E i casi i dati nulli sono troppi per essere semplicemente eliminati?

La gestione dei dati nulli è un aspetto molto complesso, **non esiste una soluzione unica ma bisogna valutare di caso in caso, a seconda dei dati che abbiamo a disposizione**, quale strategia utilizzare.

Ad esempio: se ho dei nulli in una colonna numerica, posso rimpiazzarli con la media o la mediana dei valori di quella colonna. Ma può darsi che un dato mancante stia ad indicare uno 0, e quindi scegliendo un valore statistico andiamo ad inficiare la qualità del dataset.

Può darsi troviamo i dati mancanti da altre fonti, e possiamo in questo modo generare un dataset completo utilizzando dei `.merge()` o dei `.join()`, ma molto spesso questo non accade e dobbiamo necessariamente riempire i dati nulli facendo delle assunzioni.

Dati nulli

Dopo aver trovato la logica migliore per gestire i dati nulli, Pandas ci offre dei metodi per gestire i valori nulli tramite sostituzione. Questi metodi sono:

- ❑ `.fillna()`: Riempie i valori mancanti con un valore specificato
- ❑ `.dropna()`: Elimina righe o colonne che contengono valori mancanti
- ❑ `.interpolate()`: Effettua l'interpolazione dei valori mancanti utilizzando vari metodi come lineare, polinomiale, eccetera
- ❑ `.ffill()`: Sostituisce i valori mancanti con il valore precedente nella serie
- ❑ `.bfill()`: Sostituisce i valori mancanti con il valore successivo nella serie

Dati nulli

```
temp_data = pd.DataFrame({  
    "day": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12],  
    "temperature": [20, 21, 22, None, 25, 25, None, 26, 27, 27, None, 30]  
})  
print(temp_data)
```

	day	temperature
0	1	20.0
1	2	21.0
2	3	22.0
3	4	NaN
4	5	25.0
5	6	25.0
6	7	NaN
7	8	26.0
8	9	27.0
9	10	27.0
10	11	NaN
11	12	30.0

Dati nulli

```
temp_data.fillna(0)
```

	day	temperature
0	1	20.0
1	2	21.0
2	3	22.0
3	4	0.0
4	5	25.0
5	6	25.0
6	7	0.0
7	8	26.0
8	9	27.0
9	10	27.0
10	11	0.0
11	12	30.0

Dati nulli

```
temp_data.fillna( temp_data.mean() )
```

	day	temperature
0	1	20.000000
1	2	21.000000
2	3	22.000000
3	4	24.777778
4	5	25.000000
5	6	25.000000
6	7	24.777778
7	8	26.000000
8	9	27.000000
9	10	27.000000
10	11	24.777778
11	12	30.000000

Dati nulli

```
temp_data.dropna()
```

	day	temperature
0	1	20.0
1	2	21.0
2	3	22.0
4	5	25.0
5	6	25.0
7	8	26.0
8	9	27.0
9	10	27.0
11	12	30.0

Dati nulli

```
temp_data.ffmpeg()
```

	day	temperature
0	1	20.0
1	2	21.0
2	3	22.0
3	4	22.0
4	5	25.0
5	6	25.0
6	7	25.0
7	8	26.0
8	9	27.0
9	10	27.0
10	11	27.0
11	12	30.0

Dati nulli

```
temp_data.bfill()
```

day		temperature
0	1	20.0
1	2	21.0
2	3	22.0
3	4	25.0
4	5	25.0
5	6	25.0
6	7	26.0
7	8	26.0
8	9	27.0
9	10	27.0
10	11	30.0
11	12	30.0

Dati nulli

```
temp_data.interpolate()
```

	day	temperature
0	1	20.0
1	2	21.0
2	3	22.0
3	4	23.5
4	5	25.0
5	6	25.0
6	7	25.5
7	8	26.0
8	9	27.0
9	10	27.0
10	11	28.5
11	12	30.0



GRAZIE
Epicode