Cleaning

- renommer des colonnes (df.rename(columns=....))
- trouver/supprimer les données dupliquées (.duplicated / .drop_duplicates)
- trouver les NA (df.column.isna() / df.column.notna())
- remplacer les NA (.fillna())
- remplacer n'importe quelle valeur (.replace({OLD_VALUE: NEW_VALUE, …}))
- changer le type d'une série (aka cast) (.astype(type) / pd.to_numeric / pd.to_datetime)
- fallback les valeurs NA d'une colonne sur une autre colonne: combine_first

.dt accessor (for date-type columns)

→ https://pandas.pydata.org/pandas-docs/stable/reference/series.html#datetime-properties (https://pandas.pydata.org/pandas-docs/stable/reference/series.html#datetime-properties)

.str accessor (for string-typed columns)

→ https://pandas.pydata.org/pandas-docs/stable/reference/series.html#string-handling (https://pandas.pydata.org/pandas-docs/stable/reference/series.html#string-handling)

Regexes:

- cheat sheet: https://www.debuggex.com/cheatsheet/regex/python (https://www.debuggex.com/cheatsheet/regex/python)
- talk sympa: https://www.youtube.com/watch?v=abrcJ9MpF60 (https://www.youtube.com/watch?v=abrcJ9MpF60)
- le module dédié "re" de python: https://docs.python.org/3/library/re.html (https://docs.python.org/3/library/re.html)
- site web pour tester des regexes: https://regex101.com (https://regex101.com)

Entrée [1]:

```
# if you don't want pandas.read_csv to mess with data types,
# you can force it to keep str values by specifying dtype=str.
people = pd.read_csv('people.csv')
```

Entrée [2]:

```
people.shape
```

Out[2]:

```
(209, 15)
```

Entrée [3]:

```
people.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 209 entries, 0 to 208
Data columns (total 15 columns):
id                 209 non-null int64
first_name         207 non-null object
last_name          207 non-null object
email address      203 non-null object
gender             207 non-null object
age                207 non-null object
money              190 non-null object
lon                207 non-null float64
lat                207 non-null float64
phone              83 non-null object
registration       207 non-null object
inactive           207 non-null object
last_seen          190 non-null float64
address            207 non-null object
preference         207 non-null object
dtypes: float64(3), int64(1), object(11)
memory usage: 24.6+ KB
```

Entrée [4]:

```
people.gender.unique()
```

Out[4]:

```
array(['Female', 'Male', 'F', 'M', nan], dtype=object)
```

Entrée [5]:

```
def clean_people(df):
    # rename columns:
    df = df.rename(columns={'email address': 'email'})

    # remove rows which have an empty "first_name" (NA):
    #df = df[df.first_name.notna()] <- equivalent to next line:
    df = df.dropna(subset=['first_name'])

    # drop duplicates on ID column:
    df = df.drop_duplicates()

    # Normalize gender column:
    df['gender'] = df['gender'].replace({'Female': 'F', 'Male': 'M'})

    # Convert column "age" to number (coerce: put NaN for bad values):
    df['age'] = pd.to_numeric(df.age, errors='coerce')

    # Convert columns to date type:
    df['registration'] = pd.to_datetime(df.registration)
    df['last_seen'] = pd.to_datetime(df.last_seen, unit='s')
    # When missing, last seen should fallback to the registration date:
    df['last_seen'] = df.last_seen.combine_first(df.registration)

    # Add a "full_name" column by concatenating two other ones:
    df['full_name'] = df.first_name + " " + df.last_name

    # Add a "country" column by extracting it from the address, with a split:
    df['country'] = df.address.str.split(', ').str[1]

    # Column "money" contains values like "$50.23" or "€23,09".
    # We want to make it uniform (only dollar currency) and as number, not str.
    df['currency'] = df.money.str[0]  # extract first char ($/€) to a new "currency
    df['money'] = df.money.str[1:].str.replace(',', '.')  # extract remaining chars
    df['money'] = pd.to_numeric(df.money)  # convert to number
    # convert euros cells to dollar:
    df.loc[df.currency == '€', 'money'] = df[df.currency == '€'].money * 1.10
    del df['currency']  # remove "currency" column which is now useless

    # Keep only rows where email is not NA:
    df = df.dropna(subset=['email'])
    # Keep only rows where email is a good email:
    # CAUTION: in the real world you should not use dummy regexes like this to vali
    # but instead use a dedicated tool like https://github.com/syrusakbary/validate
    df = df[df.email.str.contains('.+@[0-9a-zA-Z\.\-_]+\.\w{2,}')]
    # Some users may use email alias (example: john.smith+truc@gmail.com is an alia
    # We want to drop these duplicates. To do that, we extract the 'alias' part wit
    groups = df.email.str.extract('([0-9a-zA-Z\.\-_]+)(\+[0-9a-zA-Z\.\-_]+)?(@[0-9a
    df['email'] = groups[0] + groups[2]  # we override the email with the email wit
    # Then, just use drop_duplicates, which will keep the first line by default:
    df = df.drop_duplicates(subset=['email'])

    return df

df_clean = clean_people(people)
```

Pandas performances

- manipuler des nombres / bool est beaucoup + performant que manipuler des str
- une sélection sur un index trié est beaucoup + performant que filtrer les valeurs d'une colonne
- [https://engineering.upside.com/a-beginners-guide-to-optimizing-pandas-code-for-speed-c09ef2c6a4d6 (https://engineering.upside.com/a-beginners-guide-to-optimizing-pandas-code-for-speed-c09ef2c6a4d6)](https://engineering.upside.com/a-beginners-guide-to-optimizing-pandas-code-for-speed-c09ef2c6a4d6) : règle générale: ne pas faire de boucle for sur les lignes, et éviter le .apply autant que possible

Entrée [6]:

```
df = pd.read_csv('more_people.csv')  # big dataset
```

Entrée [7]:

```
%%timeit
# Le filtre avec .contains sur une colonne str est LENT:
x = df[df.preference.str.contains('dessert')]
```

```
999 ms ± 23.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Entrée [8]:

```
%%timeit
# Le filtre sur une colonne int est beaucoup plus rapide (presque 100x):
x = df[df.id == 27625]
```

```
15.1 ms ± 211 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Si on est amenés à faire de nombreuses fois le filtre sur préférence, pour avoir de meilleures perfs, on peut vouloir extraire les différentes valeurs (entrée/plat/dessert/boisson) dans des colonnes dédiées de type int, avec str.get_dummies

Entrée [9]:

```
df['take_dessert'] = df.preference.str.get_dummies(sep='/').dessert
df = df.set_index('take_dessert').sort_index()
```

Entrée [10]:

```
%%timeit
x = df.loc[1]
```

```
290 µs ± 8.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Interpolate

Entrée [11]:

```
temperatures = pd.DataFrame({
    'temp': [13, 14, np.nan, np.nan, 16, 17, 17, 18, 19, 19]
}, index=['6h', '7h', '8h', '9h', '10h', '11h', '12h', '13h', '14h', '15h'])
```

Entrée [12]:

```
temperatures
```

Out[12]:

	temp
6h	13.0
7h	14.0
8h	NaN
9h	NaN
10h	16.0
11h	17.0
12h	17.0
13h	18.0
14h	19.0
15h	19.0

Entrée [13]:

```
temperatures.temp.interpolate()
```

Out[13]:

```
6h     13.000000
7h     14.000000
8h     14.666667
9h     15.333333
10h    16.000000
11h    17.000000
12h    17.000000
13h    18.000000
14h    19.000000
15h    19.000000
Name: temp, dtype: float64
```

Reindex

Entrée [14]:

```
# ventes cumulées du lundi au dimanche (l'index représente le jour de la semaine):
df = pd.DataFrame({
    'total_ventes': [122, 232, 412, 598, 632]
}, index=[1, 2, 4, 5, 7])
```

Entrée [15]:

```
df
```

Out[15]:

	total_ventes
1	122
2	232
4	412
5	598
7	632

Entrée [16]:

```
df.reindex(range(1, 8))
```

Out[16]:

	total_ventes
1	122.0
2	232.0
3	NaN
4	412.0
5	598.0
6	NaN
7	632.0

Entrée [17]:

```
df.reindex(range(1, 8), method='ffill')  # compléter les NaN avec la valeur précéde
```

Out[17]:

	total_ventes
1	122
2	232
3	232
4	412
5	598
6	598
7	632

(*Note: on aurait aussi pu utiliser reindex PUIS interpolate pour compléter les NaN*)