

Nettoyage des données

(note : l'export html/pdf rend assez mal, le fichier notebook est plus joli dans la globalité. il est conseillé de lire ces notes depuis jupyter (ou autre éditeur notebook))

On commence par importer pandas puis importer les données pour nettoyer le dataset.

On se chargera uniquement de ***mettre en forme les données*** et non de les afficher dans ce fichier.

Les données nettoyées seront exportées dans un fichier csv -> ["./Datasets/MP-24-25_Cleaned.csv"](#)

```
In [2]: import pandas as pd  
dfRawPokemonData = pd.read_csv('./Datasets/MP-24-25.csv')
```

```
In [3]: dfRawPokemonData.info()
```

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 1045 entries, 0 to 1044

Data columns (total 55 columns):

#	Column	Non-Null Count	Dtype
0	Unnamed: 0.1	1045 non-null	int64
1	Unnamed: 0	1045 non-null	int64
2	pokedex_number	1045 non-null	int64
3	name	1045 non-null	object
4	german_name	1045 non-null	object
5	japanese_name	1045 non-null	object
6	is_generation_1	1045 non-null	bool
7	is_generation_2	1045 non-null	bool
8	is_generation_3	1045 non-null	bool
9	is_generation_4	1045 non-null	bool
10	is_generation_5	1045 non-null	bool
11	is_generation_6	1045 non-null	bool
12	is_generation_7	1045 non-null	bool
13	is_generation_8	1045 non-null	bool
14	status	1045 non-null	object
15	species	1045 non-null	object
16	types	1045 non-null	object
17	height	1045 non-null	object
18	weight	1045 non-null	object
19	abilities_number	1045 non-null	int64
20	ability_1	1042 non-null	object
21	ability_2	516 non-null	object
22	ability_hidden	813 non-null	object
23	total_points	1045 non-null	int64
24	hp	1045 non-null	int64
25	attack	1045 non-null	int64
26	defense	1045 non-null	int64
27	sp_attack	1045 non-null	int64
28	sp_defense	1045 non-null	int64
29	speed	1045 non-null	int64
30	catch_rate	1027 non-null	float64
31	base_friendship	930 non-null	float64
32	base_experience	925 non-null	float64
33	growth_rate	1044 non-null	object
34	egg_types	1042 non-null	object
35	percentage_male	872 non-null	float64
36	egg_cycles	1044 non-null	float64
37	against_normal	1045 non-null	float64
38	against_fire	1045 non-null	float64
39	against_water	1045 non-null	float64
40	against_electric	1045 non-null	float64
41	against_grass	1045 non-null	float64
42	against_ice	1045 non-null	float64
43	against_fight	1045 non-null	float64
44	against_poison	1045 non-null	float64
45	against_ground	1045 non-null	float64
46	against_flying	1045 non-null	float64
47	against_psychic	1045 non-null	float64
48	against_bug	1045 non-null	float64
49	against_rock	1045 non-null	float64
50	against_ghost	1045 non-null	float64
51	against_dragon	1045 non-null	float64
52	against_dark	1045 non-null	float64
53	against_steel	1045 non-null	float64
54	against_fairy	1045 non-null	float64

```
dtypes: bool(8), float64(23), int64(11), object(13)
memory usage: 392.0+ KB
```

On remarque que la plupart du set est cohérent (au niveau des nulls), mais il faut cependant retirer des colonnes qui ne nous serviront pas :

- Le nom Japonais des Pokémons
- Le nom Allemand des Pokémons
- les deux colonnes Unnamed placées au début qui servaient sûrement à la base d'index
- On pourrait aussi retirer la colonne "Base_Friendship", comme c'est une statistique qui n'est quasiment pas implémentée dans le jeu vidéo, et qui ne nous permettra donc pas d'émettre beaucoup de conclusions...

C'est parti !

```
In [5]: dfPokemonList = dfRawPokemonData.drop(columns=['german_name', 'japanese_name', 'Un
```

```
In [6]: dfPokemonList.columns
```

```
Out[6]: Index(['pokedex_number', 'name', 'is_generation_1', 'is_generation_2',
               'is_generation_3', 'is_generation_4', 'is_generation_5',
               'is_generation_6', 'is_generation_7', 'is_generation_8', 'status',
               'species', 'types', 'height', 'weight', 'abilities_number', 'ability_1',
               'ability_2', 'ability_hidden', 'total_points', 'hp', 'attack',
               'defense', 'sp_attack', 'sp_defense', 'speed', 'catch_rate',
               'base_friendship', 'base_experience', 'growth_rate', 'egg_types',
               'percentage_male', 'egg_cycles', 'against_normal', 'against_fire',
               'against_water', 'against_electric', 'against_grass', 'against_ice',
               'against_fight', 'against_poison', 'against_ground', 'against_flying',
               'against_psychic', 'against_bug', 'against_rock', 'against_ghost',
               'against_dragon', 'against_dark', 'against_steel', 'against_fairy'],
              dtype='object')
```

Nos colonnes désormais supprimées, on cherche maintenant à identifier les valeurs qui pourraient être gênantes à la manipulation des autres tableaux

on assume que toute la colonne est déjà censée avoir le même format de données

```
In [8]: dfPokemonList[dfPokemonList['height'].str[1] != '']['height']
```

```

Out[8]: 31      11'06''
        126     28'10''
        135     35'09''
        167     21'04''
        168     21'04''
        191     13'01''
        255     30'02''
        256     34'05''
        302     17'01''
        303     12'06''
        386     47'07''
        417     20'04''
        459     14'09''
        460     32'02''
        461     11'06''
        462     16'05''
        463     23'00''
        464     35'05''
        578     17'09''
        579     13'09''
        581     12'02''
        582     14'09''
        583     22'08''
        590     10'06''
        594     10'10''
        750     10'06''
        755     10'10''
        756     11'10''
        838     19'00''
        839     16'05''
        841     14'09''
        845     21'04''
        878     26'11''
        914     12'10''
        924     11'02''
        925     13'01''
        929     12'06''
        930     30'02''
        932     18'01''
        934     12'06''
        935     13'09''
        936     24'07''
        940     11'10''
        941     18'01''
        980     12'06''
        1032    65'07''
        1033    328'01''
Name: height, dtype: object

```

Même si l'argument est peu faible (malgré que suffisant), on remarque que les seuls objets de la liste qui n'ont pas d'apostrophe "" sont les valeurs avec une taille en pieds en dizaine ou centaine, ce qui confirme (en plus des 0 null dans la première étape info) que la colonne ne contient pas de valeurs erronées

En plus, les pouces sont exprimés avec une double apostrophe, on peut donc facilement le split de la façon suivante :

```
In [10]: sSplittedHeight = dfPokemonList['height'].str.split("")
```

Je le convertis en dataframe pour faire des calculs par la suite

```
In [12]: dfSplittedHeight = pd.DataFrame(sSplittedHeight.values.tolist(), index=sSplittedHeight.index)
```

```
Out[12]:
```

	0	1	2	3
--	---	---	---	---

0	2	04		
1	3	03		
2	6	07		
3	7	10		
4	2	00		
...
1040	7	03		
1041	6	07		
1042	3	07		
1043	7	10		
1044	7	10		

1045 rows × 4 columns

On supprime les colonnes vides et on renomme les restantes...

```
In [14]: dfSplittedHeight = dfSplittedHeight.drop(columns=[2,3])
```

```
In [15]: dfSplittedHeight = dfSplittedHeight.rename(columns={0:'feet',1:'inch'})
dfSplittedHeight
```

Out[15]:

	feet	inch
0	2	04
1	3	03
2	6	07
3	7	10
4	2	00
...
1040	7	03
1041	6	07
1042	3	07
1043	7	10
1044	7	10

1045 rows × 2 columns

Convertir le type des colonnes pour faire les calculs

```
In [17]: dfSplittedHeight['feet'] = dfSplittedHeight['feet'].astype('float')
dfSplittedHeight['inch'] = dfSplittedHeight['inch'].astype('float')

dfSplittedHeight.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1045 entries, 0 to 1044
Data columns (total 2 columns):
#   Column  Non-Null Count  Dtype
---  ---
0   feet    1045 non-null    float64
1   inch    1045 non-null    float64
dtypes: float64(2)
memory usage: 16.5 KB
```

et on termine ce dataset en ajoutant une colonne qui contient la taille en mètres

```
In [19]: dfSplittedHeight.insert(2, 'meters', value= dfSplittedHeight['feet'] * 0.3048 +
```

```
In [20]: dfSplittedHeight
```

Out[20]:

	feet	inch	meters
0	2.0	4.0	0.7112
1	3.0	3.0	0.9906
2	6.0	7.0	2.0066
3	7.0	10.0	2.3876
4	2.0	0.0	0.6096
...
1040	7.0	3.0	2.2098
1041	6.0	7.0	2.0066
1042	3.0	7.0	1.0922
1043	7.0	10.0	2.3876
1044	7.0	10.0	2.3876

1045 rows × 3 columns

Victoire ! on a notre taille en mètres, on remplace donc nos données dans le dataset initial.

```
In [22]: dfPokemonList['height'] = dfSplittedHeight['meters']
```

```
In [23]: dfPokemonList[['name', 'height']]
```

Out[23]:

	name	height
0	Bulbasaur	0.7112
1	Ivysaur	0.9906
2	Venusaur	2.0066
3	Mega Venusaur	2.3876
4	Charmander	0.6096
...
1040	Glastrier	2.2098
1041	Spectrier	2.0066
1042	Calyrex	1.0922
1043	Calyrex Ice Rider	2.3876
1044	Calyrex Shadow Rider	2.3876

1045 rows × 2 columns

on s'attaque au poids

```
In [25]: sSplittedWeight = dfPokemonList['weight'].str.split("lbs")
```

même stratégie que précédemment, on le convertit en df

```
In [27]: dfSplittedWeight = pd.DataFrame(sSplittedWeight.values.tolist(), index=sSplittedWeight.index)
```

```
Out[27]:
```

	0	1
0	15.21	
1	28.66	
2	220.46	
3	342.82	
4	18.74	
...
1040	1763.7	
1041	98.11	
1042	16.98	
1043	1783.76	
1044	118.17	

1045 rows × 2 columns

```
In [28]: dfSplittedWeight = dfSplittedWeight.drop(columns=[1])
```

```
In [29]: dfSplittedWeight = dfSplittedWeight.rename(columns={0: 'lbs'})
dfSplittedWeight
```


Out[29]:

	lbs
0	15.21
1	28.66
2	220.46
3	342.82
4	18.74
...	...
1040	1763.7
1041	98.11
1042	16.98
1043	1783.76
1044	118.17

1045 rows × 1 columns

Cette fois-ci, pour vérifier que la colonne ne contienne pas de valeurs erronées, et comme on ne divise qu'en une seule partie j'utilise :

```
In [31]: dfSplittedWeight[pd.to_numeric(dfSplittedWeight['lbs'],errors="coerce").isna() =
```

Out[31]:

	lbs
1033	nan

Une seule valeur null dans la liste, on regarde quel est le pokemon concerné

```
In [33]: dfPokemonList.iloc[1033]
```

```

Out[33]: pokdex_number      890
         name                Eternatus Eternamax
         is_generation_1      False
         is_generation_2      False
         is_generation_3      False
         is_generation_4      False
         is_generation_5      False
         is_generation_6      False
         is_generation_7      False
         is_generation_8      True
         status                Legendary
         species              Gigantic Pokémon
         types                Poison, Dragon
         height               99.9998
         weight               nan lbs
         abilities_number      0
         ability_1            NaN
         ability_2            NaN
         ability_hidden        NaN
         total_points          1125
         hp                   255
         attack                115
         defense               250
         sp_attack             125
         sp_defense            250
         speed                 130
         catch_rate            NaN
         base_friendship       NaN
         base_experience        NaN
         growth_rate           Slow
         egg_types             Undiscovered
         percentage_male       NaN
         egg_cycles            120.0
         against_normal        1.0
         against_fire          0.5
         against_water         0.5
         against_electric      0.5
         against_grass         0.25
         against_ice           2.0
         against_fight         0.5
         against_poison        0.5
         against_ground        2.0
         against_flying        1.0
         against_psychic       2.0
         against_bug           0.5
         against_rock          1.0
         against_ghost         1.0
         against_dragon        2.0
         against_dark          1.0
         against_steel         1.0
         against_fairy         1.0
         Name: 1033, dtype: object

```

D'après poképédia, ce pokemon ne possède en effet pas de poids. pour des raisons statistiques et de simplicité, on considèrera donc que le poids du pokemon est la moyenne des autres...

```

In [35]: EternatusEtermaxWeight = dfSplittedWeight['lbs'].drop(index=[1033]).astype('float)

```

```
In [36]: print(EternatusEtermaxWeight)
```

157.00548850574714

Sa valeur attribuée est donc 157.00548850574714.

```
In [38]: dfSplittedWeight.loc[1033, 'lbs'] = EternatusEtermaxWeight
```

Convertir le type des colonnes pour faire les calculs

```
In [40]: dfSplittedWeight['lbs'] = dfSplittedWeight['lbs'].astype('float')

dfSplittedWeight.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1045 entries, 0 to 1044
Data columns (total 1 columns):
#   Column  Non-Null Count  Dtype  
---  -
0    lbs      1045 non-null     float64
dtypes: float64(1)
memory usage: 8.3 KB
```

on crée donc notre conversion dans le df du poids

```
In [42]: dfSplittedWeight.insert(1, 'kg' , value= 0.454 * dfSplittedWeight['lbs'], allow_
```

```
In [43]: dfSplittedWeight
```

```
Out[43]:
```

	lbs	kg
0	15.21	6.90534
1	28.66	13.01164
2	220.46	100.08884
3	342.82	155.64028
4	18.74	8.50796
...
1040	1763.70	800.71980
1041	98.11	44.54194
1042	16.98	7.70892
1043	1783.76	809.82704
1044	118.17	53.64918

1045 rows × 2 columns

Le résultat n'est pas arrondi mais on peut quand même le mettre dans le dataset initial.

```
In [45]: dfPokemonList['weight'] = dfSplittedWeight['kg']
```

```
In [46]: dfPokemonList[['name','height','weight']]
```

```
Out[46]:
```

	name	height	weight
0	Bulbasaur	0.7112	6.90534
1	Ivysaur	0.9906	13.01164
2	Venusaur	2.0066	100.08884
3	Mega Venusaur	2.3876	155.64028
4	Charmander	0.6096	8.50796
...
1040	Glastrier	2.2098	800.71980
1041	Spectrier	2.0066	44.54194
1042	Calyrex	1.0922	7.70892
1043	Calyrex Ice Rider	2.3876	809.82704
1044	Calyrex Shadow Rider	2.3876	53.64918

1045 rows × 3 columns

Maintenant, on cherche à remplacer nos colonnes générations par une colonne unique qui représente la génération par un entier plutôt que 8 booléens

(la solution présentée suivante a été trouvée sur stackoverflow : [Pandas multiple boolean columns to new single column with mapping dict](#))

```
In [48]: dfGenerationColumns = dfPokemonList.filter(like='generation_')
```

```
In [49]: dfGenerationColumns
```

```
Out[49]:
```

	is_generation_1	is_generation_2	is_generation_3	is_generation_4	is_generation_5
0	True	False	False	False	False
1	True	False	False	False	False
2	True	False	False	False	False
3	True	False	False	False	False
4	True	False	False	False	False
...
1040	False	False	False	False	False
1041	False	False	False	False	False
1042	False	False	False	False	False
1043	False	False	False	False	False
1044	False	False	False	False	False

1045 rows × 8 columns

```
In [50]: mapping_dict = {False: "none", "is_generation_1": "1", "is_generation_2": "2", "
```

On utilise ce sous ensemble et ce dictionnaire pour mapper nos colonnes en une seule nouvelle colonne :

```
In [52]: dfGenerationColumns.insert(allow_duplicates=True, column='generation', loc=8, value
```

```
In [53]: dfGenerationColumns['generation'] = (dfGenerationColumns.idxmax(axis=1).map(mapp
```

C:\Users\Aubin\AppData\Local\Temp\ipykernel_11108\3306450642.py:1: SettingWithCopyWarning:

A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy

```
dfGenerationColumns['generation'] = (dfGenerationColumns.idxmax(axis=1).map(mapping_dict).where(dfGenerationColumns.any(axis=1), mapping_dict[False]))
```

On insère notre colonne calculée à notre dataset initial...

```
In [55]: dfPokemonList.insert(column='generation', loc=2, value=dfGenerationColumns['gene
```

```
In [56]: dfPokemonList[dfPokemonList['generation'] == 0]
```

```
Out[56]:
```

pokedex_number	name	generation	is_generation_1	is_generation_2	is_generation_3
----------------	------	------------	-----------------	-----------------	-----------------

0 rows × 52 columns

On voit que notre dataset ne contient pas de lignes avec generation = 0, donc nos générations on été mappées sans erreurs, on prend aussi un exemple pour faire une

double vérification :

```
In [58]: dfPokemonList.iloc[430] # nombre choisi aléatoirement
```

```
Out[58]: pokedex_number      359
name          Absol
generation      3
is_generation_1  False
is_generation_2  False
is_generation_3   True
is_generation_4  False
is_generation_5  False
is_generation_6  False
is_generation_7  False
is_generation_8  False
status          Normal
species          Disaster Pokémon
types            Dark
height          1.1938
weight          47.04348
abilities_number      3
ability_1        Pressure
ability_2        Super Luck
ability_hidden    Justified
total_points      465
hp               65
attack           130
defense           60
sp_attack         75
sp_defense        60
speed             75
catch_rate        30.0
base_friendship   35.0
base_experience   163.0
growth_rate       Medium Slow
egg_types          Field
percentage_male    50.0
egg_cycles         25.0
against_normal     1.0
against_fire       1.0
against_water      1.0
against_electric   1.0
against_grass      1.0
against_ice        1.0
against_fight      2.0
against_poison     1.0
against_ground     1.0
against_flying     1.0
against_psychic    0.0
against_bug        2.0
against_rock       1.0
against_ghost      0.5
against_dragon     1.0
against_dark       0.5
against_steel      1.0
against_fairy      2.0
Name: 430, dtype: object
```

Nos données correspondent, OK.

On va retirer les anciennes colonnes

```
In [60]: dfPokemonList = dfPokemonList.drop(columns=['is_generation_1','is_generation_2'],
```

```
In [61]: dfPokemonList
```

```
Out[61]:
```

	pokedex_number	name	generation	status	species	types	height
0	1	Bulbasaur	1	Normal	Seed Pokémon	Grass, Poison	0.7112
1	2	Ivysaur	1	Normal	Seed Pokémon	Grass, Poison	0.9906
2	3	Venusaur	1	Normal	Seed Pokémon	Grass, Poison	2.0066
3	3	Mega Venusaur	1	Normal	Seed Pokémon	Grass, Poison	2.3876
4	4	Charmander	1	Normal	Lizard Pokémon	Fire	0.6096
...
1040	896	Glastrier	8	Sub Legendary	Wild Horse Pokémon	Ice	2.2098
1041	897	Spectrier	8	Sub Legendary	Swift Horse Pokémon	Ghost	2.0066
1042	898	Calyrex	8	Legendary	King Pokémon	Psychic, Grass	1.0922
1043	898	Calyrex Ice Rider	8	Legendary	High King Pokémon	Psychic, Ice	2.3876
1044	898	Calyrex Shadow Rider	8	Legendary	High King Pokémon	Psychic, Ghost	2.3876

1045 rows × 44 columns

Après avoir nettoyé la variable génération, on va maintenant s'occuper des types ; on prend la même approche que pour la taille et pour la masse :

on commence par découper les types par leur virgule (présente ou non)

```
In [63]: sSplittedPokemonTypes = dfPokemonList['types'].str.split(",")
```

```
In [64]: sSplittedPokemonTypes[sSplittedPokemonTypes.isna()]
```

```
Out[64]: Series([], Name: types, dtype: object)
```

```
In [65]: sSplittedPokemonTypes
```

```

Out[65]: 0      [Grass, Poison]
         1      [Grass, Poison]
         2      [Grass, Poison]
         3      [Grass, Poison]
         4      [Fire]
         ...
        1040      [Ice]
        1041      [Ghost]
        1042  [Psychic, Grass]
        1043  [Psychic, Ice]
        1044  [Psychic, Ghost]
        Name: types, Length: 1045, dtype: object

```

Pas de NaN à l'horizon, on peut donc passer à la suite

On transforme la series en dataframe et on supprime les espaces qui sont venus se glisser dans les valeurs

```

In [67]: dfPokemonTypes = pd.DataFrame(sSplittedPokemonTypes.values.tolist(), index=sSplittedPokemonTypes.index)
dfPokemonTypes[1] = dfPokemonTypes[1].str.strip()
dfPokemonTypes

```

```

Out[67]:

```

	0	1
0	Grass	Poison
1	Grass	Poison
2	Grass	Poison
3	Grass	Poison
4	Fire	None
...
1040	Ice	None
1041	Ghost	None
1042	Psychic	Grass
1043	Psychic	Ice
1044	Psychic	Ghost

1045 rows × 2 columns

on compte le nombre de valeurs en l'ajoutant dans une colonne

```

In [69]: dfPokemonTypes.insert(loc=0, column='type_number', value=dfPokemonTypes.count(axis=1))

```

```

In [70]: dfPokemonTypes

```



```
Out[70]:
```

	type_number	0	1
0	2	Grass	Poison
1	2	Grass	Poison
2	2	Grass	Poison
3	2	Grass	Poison
4	1	Fire	None
...
1040	1	Ice	None
1041	1	Ghost	None
1042	2	Psychic	Grass
1043	2	Psychic	Ice
1044	2	Psychic	Ghost

1045 rows × 3 columns

maintenant qu'on a toutes les valeurs qui nous intéressent, on va modifier les noms de colonne et les agréger à notre dataframe principal

```
In [72]: dfPokemonTypes = dfPokemonTypes.rename(columns={0:'type_1',1:'type_2'})
```

```
In [73]: dfPokemonTypes
```

```
Out[73]:
```

	type_number	type_1	type_2
0	2	Grass	Poison
1	2	Grass	Poison
2	2	Grass	Poison
3	2	Grass	Poison
4	1	Fire	None
...
1040	1	Ice	None
1041	1	Ghost	None
1042	2	Psychic	Grass
1043	2	Psychic	Ice
1044	2	Psychic	Ghost

1045 rows × 3 columns

```
In [74]: dfPokemonList.info()
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1045 entries, 0 to 1044
Data columns (total 44 columns):
#   Column                Non-Null Count  Dtype
---  -
0   pokedex_number        1045 non-null   int64
1   name                  1045 non-null   object
2   generation            1045 non-null   int32
3   status               1045 non-null   object
4   species              1045 non-null   object
5   types                1045 non-null   object
6   height               1045 non-null   float64
7   weight               1045 non-null   float64
8   abilities_number     1045 non-null   int64
9   ability_1            1042 non-null   object
10  ability_2            516 non-null    object
11  ability_hidden       813 non-null    object
12  total_points         1045 non-null   int64
13  hp                   1045 non-null   int64
14  attack               1045 non-null   int64
15  defense              1045 non-null   int64
16  sp_attack            1045 non-null   int64
17  sp_defense           1045 non-null   int64
18  speed                1045 non-null   int64
19  catch_rate           1027 non-null   float64
20  base_friendship      930 non-null    float64
21  base_experience       925 non-null    float64
22  growth_rate          1044 non-null   object
23  egg_types            1042 non-null   object
24  percentage_male      872 non-null    float64
25  egg_cycles           1044 non-null   float64
26  against_normal       1045 non-null   float64
27  against_fire         1045 non-null   float64
28  against_water        1045 non-null   float64
29  against_electric     1045 non-null   float64
30  against_grass        1045 non-null   float64
31  against_ice          1045 non-null   float64
32  against_fight        1045 non-null   float64
33  against_poison       1045 non-null   float64
34  against_ground       1045 non-null   float64
35  against_flying       1045 non-null   float64
36  against_psychic      1045 non-null   float64
37  against_bug          1045 non-null   float64
38  against_rock         1045 non-null   float64
39  against_ghost        1045 non-null   float64
40  against_dragon       1045 non-null   float64
41  against_dark         1045 non-null   float64
42  against_steel        1045 non-null   float64
43  against_fairy        1045 non-null   float64
dtypes: float64(25), int32(1), int64(9), object(9)
memory usage: 355.3+ KB

```

avec toutes nos modifications, on voit que le type est le numéro de colonne '5', on va donc insérer notre dataframe à la colonne 5, puis supprimer l'ancienne.

```

In [76]: dfPokemonList.insert(loc=5, column='type_number', value= dfPokemonTypes['type_nu
dfPokemonList.insert(loc=6, column='type_1', value= dfPokemonTypes['type_1'], al
dfPokemonList.insert(loc=7, column='type_2', value= dfPokemonTypes['type_2'], al

```

```
In [77]: dfPokemonList = dfPokemonList.drop(columns='types')
```

```
In [78]: dfPokemonList
```

```
Out[78]:
```

	pokedex_number	name	generation	status	species	type_number	ty
0	1	Bulbasaur	1	Normal	Seed Pokémon	2	(
1	2	Ivysaur	1	Normal	Seed Pokémon	2	(
2	3	Venusaur	1	Normal	Seed Pokémon	2	(
3	3	Mega Venusaur	1	Normal	Seed Pokémon	2	(
4	4	Charmander	1	Normal	Lizard Pokémon	1	
...	
1040	896	Glastrier	8	Sub Legendary	Wild Horse Pokémon	1	
1041	897	Spectrier	8	Sub Legendary	Swift Horse Pokémon	1	(
1042	898	Calyrex	8	Legendary	King Pokémon	2	Ps
1043	898	Calyrex Ice Rider	8	Legendary	High King Pokémon	2	Ps
1044	898	Calyrex Shadow Rider	8	Legendary	High King Pokémon	2	Ps

1045 rows × 46 columns

On fait la même chose pour le types d'oeufs :

```
In [80]: sSplittedEggTypes = dfPokemonList['egg_types'].str.split(",")
```

```
In [81]: sSplittedEggTypes[sSplittedEggTypes.isna()]
```

```
Out[81]: 33      NaN
172      NaN
658      NaN
Name: egg_types, dtype: object
```

On remarque que certains pokemons n'ont pas de types d'oeufs. après vérification (non montrée ici), les trois pokemons concernés sont : Pikachu Partenaire, Evoli Partenaire, et Darumarond de galar (mode Transe). Pour ne pas se séparer de ces lignes, et comme ce sont des pokemons qui ont d'autres formes, on peut facilement remplacer ces valeurs par

la mode des autres.

On remplacera donc ici, par la valeur des pokémons dans leur version standard.

On cherche donc nos types d'oeufs

```
In [83]: dfPokemonList[dfPokemonList['name'] == 'Pikachu']['name','egg_types']
```

```
Out[83]:
```

	name	egg_types
32	Pikachu	Fairy, Field

```
In [84]: dfPokemonList[dfPokemonList['name'] == 'Eevee']['name','egg_types']
```

```
Out[84]:
```

	name	egg_types
171	Eevee	Field

```
In [85]: dfPokemonList[dfPokemonList['name'] == 'Galarian Darmanitan Standard Mode']['na
```

```
Out[85]:
```

	name	egg_types
657	Galarian Darmanitan Standard Mode	Field

d'après notre liste déjà existante, on sait donc que les types d'oeufs sont :

- Pikachu : Field, Fairy
- Eevee : Field
- Galarian Darmanitan Standard Mode : Field

on remplace nos valeurs manquantes.

```
In [87]: sSplittedEggTypes.iloc[33] = ['Fairy','Field'] # Pikachu  
sSplittedEggTypes.iloc[172] = ['Field'] # Eevee  
sSplittedEggTypes.iloc[658] = ['Field'] # Darmanitan
```

```
In [88]: sSplittedEggTypes[sSplittedEggTypes.isna()]
```

```
Out[88]: Series([], Name: egg_types, dtype: object)
```

Plus de NaN ! , on va pouvoir finir nos opérations

```
In [90]: dfEggTypes = pd.DataFrame(sSplittedEggTypes.values.tolist(), index=sSplittedEggT  
dfEggTypes
```

```
Out[90]:
```

	0	1
0	Grass	Monster
1	Grass	Monster
2	Grass	Monster
3	Grass	Monster
4	Dragon	Monster
...
1040	Undiscovered	None
1041	Undiscovered	None
1042	Undiscovered	None
1043	Undiscovered	None
1044	Undiscovered	None

1045 rows × 2 columns

```
In [91]: dfEggTypes.insert(loc=0, column='egg_type_number', value=dfEggTypes.count(axis=1
```

```
In [92]: dfEggTypes
```

```
Out[92]:
```

	egg_type_number	0	1
0	2	Grass	Monster
1	2	Grass	Monster
2	2	Grass	Monster
3	2	Grass	Monster
4	2	Dragon	Monster
...
1040	1	Undiscovered	None
1041	1	Undiscovered	None
1042	1	Undiscovered	None
1043	1	Undiscovered	None
1044	1	Undiscovered	None

1045 rows × 3 columns

On renomme nos colonnes :

```
In [94]: dfEggTypes = dfEggTypes.rename(columns={0:'egg_type_1',1:'egg_type_2'})
```

Il existe des Pokemon qui n'ont pas de type d'oeufs decouverts, on remplacera donc leur

nombre de type par '0' dans ces cas là.

```
In [96]: dfEggTypes.loc[dfEggTypes['egg_type_1'] == 'Undiscovered', 'egg_type_number'] =
```

```
In [97]: dfEggTypes[dfEggTypes['egg_type_1'] == 'Undiscovered']
```

```
Out[97]:
```

	egg_type_number	egg_type_1	egg_type_2
41	0	Undiscovered	None
42	0	Undiscovered	None
184	0	Undiscovered	None
185	0	Undiscovered	None
186	0	Undiscovered	None
...
1040	0	Undiscovered	None
1041	0	Undiscovered	None
1042	0	Undiscovered	None
1043	0	Undiscovered	None
1044	0	Undiscovered	None

152 rows × 3 columns

OK.

Comme pour les types de pokemon, on agrège à notre data frame principal. On cherche le numéro de colonne pour l'insérer dans le dataframe.

```
In [99]: dfPokemonList.info()
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1045 entries, 0 to 1044
Data columns (total 46 columns):
#   Column                Non-Null Count  Dtype
---  -
0   pokedex_number        1045 non-null   int64
1   name                  1045 non-null   object
2   generation            1045 non-null   int32
3   status                1045 non-null   object
4   species               1045 non-null   object
5   type_number           1045 non-null   int64
6   type_1                1045 non-null   object
7   type_2                553 non-null    object
8   height                1045 non-null   float64
9   weight                1045 non-null   float64
10  abilities_number      1045 non-null   int64
11  ability_1             1042 non-null   object
12  ability_2             516 non-null    object
13  ability_hidden        813 non-null    object
14  total_points          1045 non-null   int64
15  hp                    1045 non-null   int64
16  attack                1045 non-null   int64
17  defense               1045 non-null   int64
18  sp_attack             1045 non-null   int64
19  sp_defense            1045 non-null   int64
20  speed                 1045 non-null   int64
21  catch_rate            1027 non-null   float64
22  base_friendship       930 non-null    float64
23  base_experience        925 non-null    float64
24  growth_rate           1044 non-null   object
25  egg_types              1042 non-null   object
26  percentage_male        872 non-null    float64
27  egg_cycles            1044 non-null   float64
28  against_normal        1045 non-null   float64
29  against_fire          1045 non-null   float64
30  against_water         1045 non-null   float64
31  against_electric      1045 non-null   float64
32  against_grass         1045 non-null   float64
33  against_ice           1045 non-null   float64
34  against_fight         1045 non-null   float64
35  against_poison        1045 non-null   float64
36  against_ground        1045 non-null   float64
37  against_flying        1045 non-null   float64
38  against_psychic       1045 non-null   float64
39  against_bug           1045 non-null   float64
40  against_rock          1045 non-null   float64
41  against_ghost         1045 non-null   float64
42  against_dragon        1045 non-null   float64
43  against_dark          1045 non-null   float64
44  against_steel         1045 non-null   float64
45  against_fairy         1045 non-null   float64
dtypes: float64(25), int32(1), int64(10), object(10)
memory usage: 371.6+ KB

```

loc à insérer : 25

In [101...

```

dfPokemonList.insert(loc=25, column='egg_type_number', value= dfEggTypes['egg_ty
dfPokemonList.insert(loc=26, column='egg_type_1', value= dfEggTypes['egg_type_1'
dfPokemonList.insert(loc=27, column='egg_type_2', value= dfEggTypes['egg_type_2'

```

In [102... dfPokemonList.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1045 entries, 0 to 1044
Data columns (total 49 columns):
#   Column                Non-Null Count  Dtype
---  -
0   pokedex_number        1045 non-null  int64
1   name                  1045 non-null  object
2   generation            1045 non-null  int32
3   status                1045 non-null  object
4   species               1045 non-null  object
5   type_number           1045 non-null  int64
6   type_1                1045 non-null  object
7   type_2                553 non-null   object
8   height                1045 non-null  float64
9   weight                1045 non-null  float64
10  abilities_number      1045 non-null  int64
11  ability_1             1042 non-null  object
12  ability_2             516 non-null   object
13  ability_hidden        813 non-null   object
14  total_points          1045 non-null  int64
15  hp                    1045 non-null  int64
16  attack                1045 non-null  int64
17  defense               1045 non-null  int64
18  sp_attack             1045 non-null  int64
19  sp_defense            1045 non-null  int64
20  speed                 1045 non-null  int64
21  catch_rate            1027 non-null  float64
22  base_friendship       930 non-null   float64
23  base_experience        925 non-null   float64
24  growth_rate           1044 non-null  object
25  egg_type_number       1045 non-null  int64
26  egg_type_1            1045 non-null  object
27  egg_type_2            286 non-null   object
28  egg_types              1042 non-null  object
29  percentage_male        872 non-null   float64
30  egg_cycles            1044 non-null  float64
31  against_normal         1045 non-null  float64
32  against_fire           1045 non-null  float64
33  against_water          1045 non-null  float64
34  against_electric       1045 non-null  float64
35  against_grass          1045 non-null  float64
36  against_ice            1045 non-null  float64
37  against_fight          1045 non-null  float64
38  against_poison         1045 non-null  float64
39  against_ground         1045 non-null  float64
40  against_flying         1045 non-null  float64
41  against_psychic        1045 non-null  float64
42  against_bug            1045 non-null  float64
43  against_rock           1045 non-null  float64
44  against_ghost          1045 non-null  float64
45  against_dragon         1045 non-null  float64
46  against_dark           1045 non-null  float64
47  against_steel          1045 non-null  float64
48  against_fairy          1045 non-null  float64
dtypes: float64(25), int32(1), int64(11), object(12)
memory usage: 396.1+ KB
```


On supprime notre colonne egg_types du dataset initial

```
In [104...] dfPokemonList = dfPokemonList.drop(columns='egg_types')
```

```
In [105...] dfPokemonList
```

```
Out[105...]
   pokedex_number  name  generation  status  species  type_number  ty
0                1  Bulbasaur         1  Normal  Seed Pokémon      2  (
1                2   Ivysaur         1  Normal  Seed Pokémon      2  (
2                3  Venusaur         1  Normal  Seed Pokémon      2  (
3                3   Mega Venusaur         1  Normal  Seed Pokémon      2  (
4                4  Charmander         1  Normal  Lizard Pokémon      1
...             ...      ...      ...      ...      ...      ...
1040            896   Glastrier         8  Sub-Legendary  Wild Horse Pokémon      1
1041            897   Spectrier         8  Sub-Legendary  Swift Horse Pokémon      1  C
1042            898    Calyrex         8  Legendary      King Pokémon      2  Ps
1043            898  Calyrex Ice Rider         8  Legendary      High King Pokémon      2  Ps
1044            898  Calyrex Shadow Rider         8  Legendary      High King Pokémon      2  Ps
```

1045 rows × 48 columns

Lors de notre analyse, on s'est aperçus que la colonne 'against_ice' contenait des valeurs mal rédigées, on va donc les régler ici.

En effet, les faiblesses de pokémon ne peuvent être situées qu'entre 0 et 4, or, on a découvert que des valeurs erronées à 125 sont dans le dataset.

Ces valeurs ne sont pas impossibles, simplement mal formatées, au lieu de 125, la résistance est en réalité 0.125 (1/8, grâce à un talent).

On va d'abord identifier les pokémons touchés puis les modifier

```
In [107... print(dfPokemonList[dfPokemonList['against_ice'] > 4][['pokedex_number','name']]
dfPokemonList.loc[dfPokemonList['against_ice'] > 4, 'against_ice'] = 0.125
```

	pokedex_number	name
115	87	Dewgong
436	363	Spheal
437	364	Sealeo
438	365	Walrein

Terminé de ce côté !

On supprime le mot Pokémon de chaque espèce, pour simplement avoir le nom réel de l'espèce :

```
In [109... dfPokemonList['species'] = dfPokemonList['species'].str.replace(r' Pokémon', '',
```

Maintenant que l'on a entièrement nettoyé notre dataset, on l'enregistre en csv à l'adresse du début !

```
In [111... dfPokemonList.to_csv('./Datasets/MP-24-25_Cleaned.csv')
```

Le notebook pour le nettoyage se termine ici, pour la visualisation et l'analyse de données, on utilisera l'autre notebook, pour bien séparer les usages.

Analyse Exploratoire des Pokémons

(note : l'export html/pdf rend assez mal, le fichier notebook est plus joli dans la globalité. il est conseillé de lire ces notes depuis jupyter (ou autre éditeur notebook))

(Pour le nettoyage des données, se référer au notebook [Data Cleaning](#) (./Data_cleaning.ipynb), Ce fichier ne s'occupera que de l'analyse du jeu de données.)

Pour commencer, on va d'abord importer toutes les librairies dont nous aurons besoin lors de cette analyse :

In [240...

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from scipy.stats import skew
from scipy.stats import kurtosis
from scipy.stats import gaussian_kde
from scipy.stats import pearsonr
from scipy.stats import linregress
from scipy.stats import chi2_contingency
from scipy.stats import ttest_ind
from sklearn.preprocessing import LabelEncoder
from IPython.display import display, HTML
from scipy.stats import f_oneway
from statsmodels.stats.multicomp import pairwise_tukeyhsd
import itertools
```

Problématique

Voici la problématique générale à laquelle nous souhaitons répondre dans cette analyse :

Existe-t-il des profils types de Pokémon selon leur catégorie ou leur type ?

Cette question sera découpée en plusieurs morceaux afin de faire des sous-problématiques plus ciblées, qui pourront nous éclaircir ou nous donner des parts de réponses.

Par exemple : **Les pokémons de type Dragon sont-ils statistiquement plus puissants que les autres types ?** (et autres...)

On cherchera surtout des catégories qui sortent du lot, ou des **Plus fort / Moins Fort** (en termes de variables de combat, puissance, etc...)

On commence avant tout par ouvrir le fichier csv avec pandas :

```
In [4]: dfPokemon = pd.read_csv('./Datasets/MP-24-25_Cleaned.csv')
```

```
In [5]: dfPokemon.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1045 entries, 0 to 1044
Data columns (total 49 columns):
#   Column                Non-Null Count  Dtype
---  -
0   Unnamed: 0            1045 non-null   int64
1   pokedex_number        1045 non-null   int64
2   name                  1045 non-null   object
3   generation            1045 non-null   int64
4   status                1045 non-null   object
5   species               1045 non-null   object
6   type_number           1045 non-null   int64
7   type_1                1045 non-null   object
8   type_2                553 non-null    object
9   height                1045 non-null   float64
10  weight                1045 non-null   float64
11  abilities_number      1045 non-null   int64
12  ability_1             1042 non-null   object
13  ability_2             516 non-null    object
14  ability_hidden        813 non-null    object
15  total_points          1045 non-null   int64
16  hp                    1045 non-null   int64
17  attack                1045 non-null   int64
18  defense               1045 non-null   int64
19  sp_attack             1045 non-null   int64
20  sp_defense            1045 non-null   int64
21  speed                 1045 non-null   int64
22  catch_rate            1027 non-null   float64
23  base_friendship       930 non-null    float64
24  base_experience        925 non-null    float64
25  growth_rate           1044 non-null   object
26  egg_type_number       1045 non-null   int64
27  egg_type_1            1045 non-null   object
28  egg_type_2            286 non-null    object
29  percentage_male        872 non-null    float64
30  egg_cycles            1044 non-null   float64
31  against_normal        1045 non-null   float64
32  against_fire          1045 non-null   float64
33  against_water         1045 non-null   float64
34  against_electric      1045 non-null   float64
35  against_grass         1045 non-null   float64
36  against_ice           1045 non-null   float64
37  against_fight         1045 non-null   float64
38  against_poison        1045 non-null   float64
39  against_ground        1045 non-null   float64
40  against_flying        1045 non-null   float64
41  against_psychic       1045 non-null   float64
42  against_bug           1045 non-null   float64
43  against_rock          1045 non-null   float64
44  against_ghost         1045 non-null   float64
45  against_dragon        1045 non-null   float64
46  against_dark          1045 non-null   float64
47  against_steel         1045 non-null   float64
48  against_fairy         1045 non-null   float64
dtypes: float64(25), int64(13), object(11)
memory usage: 400.2+ KB
```

La colonne 'Unnamed: 0' s'est glissé dans notre dataset, avant de continuer notre analyse,

on va simplement retirer cette colonne :

```
In [7]: dfPokemon = dfPokemon.drop(columns='Unnamed: 0')
```

Définition des variables qui nous intéresseront dans notre étude, et Quantitatives/Qualitatives

On ignorera les variables :

- pokedex_number (simple index qui augmente, ça n'apporte pas d'intérêt de l'analyser)
- name (même raisonnement, la colonne n'apporte pas d'éléments statistiques)

ce sont quand même deux colonnes que l'on gardera sans analyser pour les différencier, car certains pokémons existent sous plusieurs formes, et l'index du tableau ne reflète pas celui du numéro de pokédex. (identification)

- abilities_number
- ability_1
- ability_2
- hidden_ability
- base_friendship
- growth_rate
- egg_types_number
- egg_type_1
- egg_type_2
- percentage_male
- egg_cycles
- against_*

On pourra peut être se servir des variables ***height et weight***, pour voir si le poids influe ou non sur certaines performances.

même si l'analyse pourrait être très intéressante, on ne considèrera pas les données sur les oeufs ici.

Pour le reste des variables :

1. Qualitatives :

- status (Normal, demi-légendaire, mythique, etc.)
- species
- type_1
- type_2

- generation

2. Quantitatives:

- height
- weight (potentiellement utilisables)
- total_points
- hp
- attack
- defense
- sp_attack
- sp_defense
- speed
- catch_rate
- base_experience

Analyse univariée

Analyse des variables quantitatives

Notre étude débutera donc avec une analyse univariée des variables quantitatives (utiles à notre exploration) de notre dataset nettoyé :

Pour rappel, la signification des variables :

- Count : le compte total de valeurs non nulles dans la colonne
- mean : la moyenne des valeurs de la colonne
- 25, 50, 75 : Valeur au 25ème centile, Valeur médiane et Valeur au 75ème centile
- min, max : Valeurs minimales et maximales de la colonne
- Kurtosis : Dispersion des données (Plutôt en pic ou plat ?)
- Skewness : Distribution des données (plutôt penchée à gauche ou à droite ?)

Les boxplots (boîtes à moustaches) fonctionnent de la manière suivante :

Les moustaches représentent le maximum et minimum, que l'on peut retrouver dans la description au dessus.

La boîte de couleur représente toutes les valeurs entre le 25ème et 75ème centile, comme mentionné précédemment.

La ligne dans la boîte de couleur (généralement orange) représente la valeur médiane.

Les variables de statistiques auront droit à une analyse particulière pour leur analyse, et seront donc exclues de cette mini analyse, on fera une observation de ces variables avant de passer aux stats

```
In [10]: univarColumnList = ['height', 'weight', 'total_points', 'catch_rate', 'base_experience']
Colors = sns.color_palette('Set1')

for index, column in enumerate(univarColumnList):
    stats = dfPokemon[column].describe()
    stats['kurtosis'] = dfPokemon[column].kurt()
    stats['skewness'] = dfPokemon[column].skew()

    html_block = f"""
    <div style="border:1px solid #ddd; padding:12px; margin-top:20px; margin-bottom:20px;">
    <h3 style="margin-top:0;">Analyse univariée : <i>{column}</i></h3>
    <ul style="margin-left:15px;">
    <li><b>Count</b> : {stats['count']:.0f}</li>
    <li><b>Mean</b> : {stats['mean']:.2f}</li>
    <li><b>Std</b> : {stats['std']:.2f}</li>
    <li><b>Min</b> : {stats['min']:.2f}</li>
    <li><b>25%</b> : {stats['25%']:.2f}</li>
    <li><b>50% (Median)</b> : {stats['50%']:.2f}</li>
    <li><b>75%</b> : {stats['75%']:.2f}</li>
    """
```



```

<li><b>Max</b> : {stats['max']:.2f}</li>
<li><b>Skewness</b> : {stats['skewness']:.2f}</li>
<li><b>Kurtosis</b> : {stats['kurtosis']:.2f}</li>
</ul>
</div>
"""
display(HTML(html_block))

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

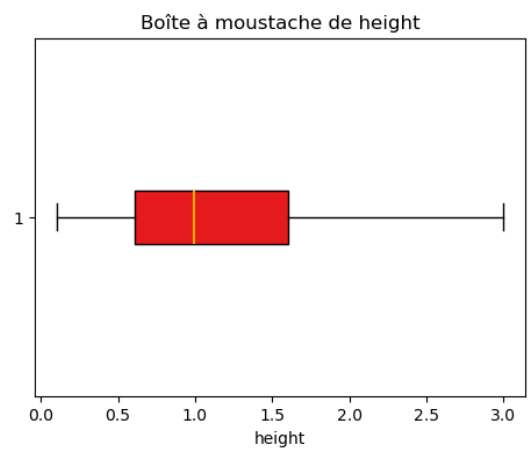
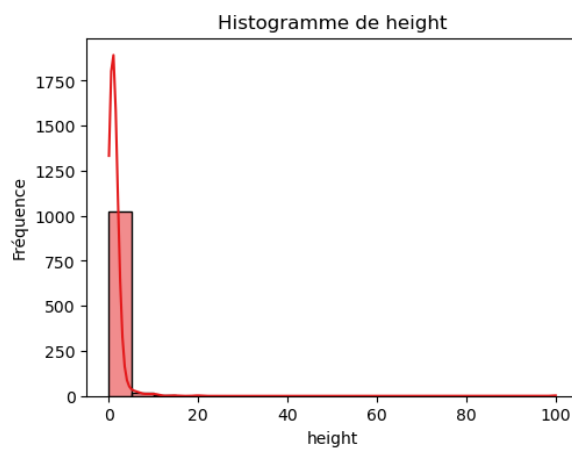
sns.histplot(dfPokemon[column].dropna(), bins=20, kde=True, color=Colors[ind
axes[0].set_title(f"Histogramme de {column}")
axes[0].set_xlabel(column)
axes[0].set_ylabel("Fréquence")

axes[1].boxplot(dfPokemon[column].dropna(), vert=False, showfliers=False, pa
axes[1].set_title(f"Boîte à moustache de {column}")
axes[1].set_xlabel(column)
plt.show()

```

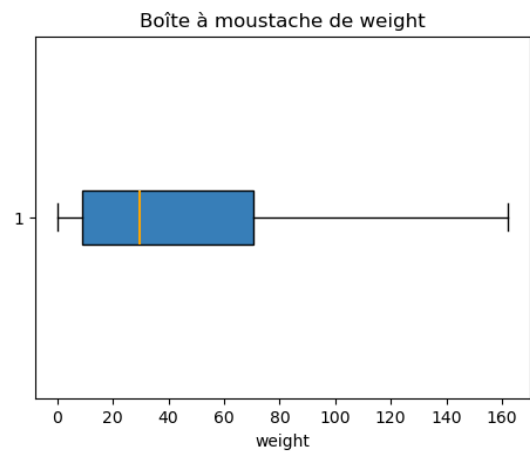
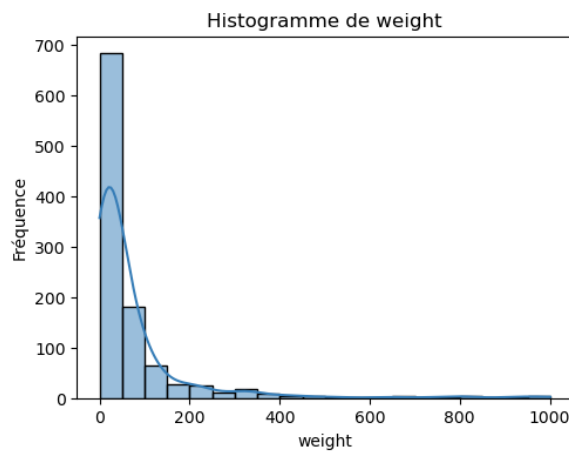
Analyse univariée : *height*

- **Count** : 1045
- **Mean** : 1.37
- **Std** : 3.35
- **Min** : 0.10
- **25%** : 0.61
- **50% (Median)** : 0.99
- **75%** : 1.60
- **Max** : 100.00
- **Skewness** : 24.81
- **Kurtosis** : 719.71



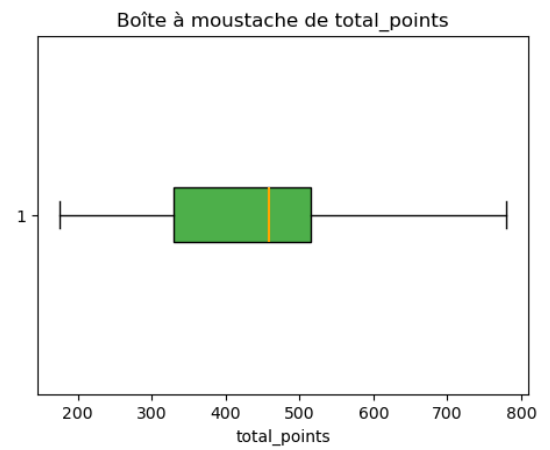
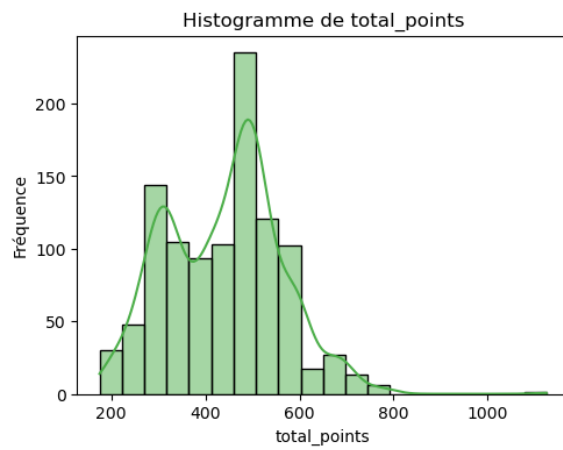
Analyse univariée : *weight*

- **Count** : 1045
- **Mean** : 71.28
- **Std** : 132.32
- **Min** : 0.10
- **25%** : 9.01
- **50% (Median)** : 29.53
- **75%** : 70.57
- **Max** : 1000.80
- **Skewness** : 4.22
- **Kurtosis** : 21.32



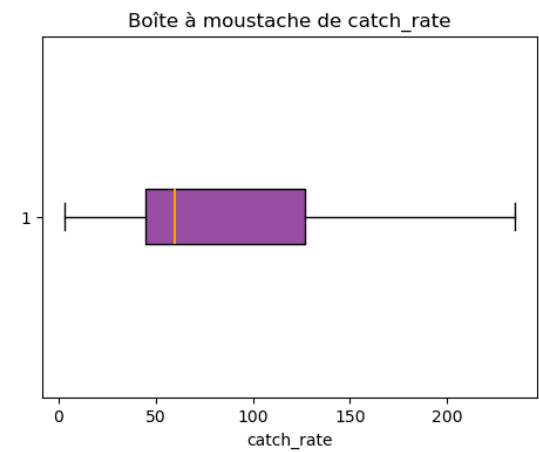
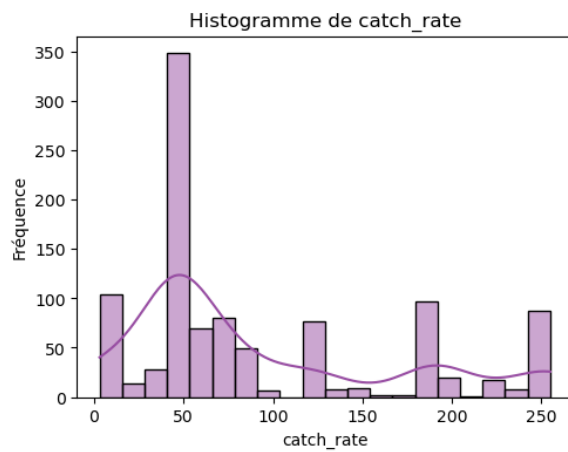
Analyse univariée : *total_points*

- **Count** : 1045
- **Mean** : 439.35
- **Std** : 121.99
- **Min** : 175.00
- **25%** : 330.00
- **50% (Median)** : 458.00
- **75%** : 515.00
- **Max** : 1125.00
- **Skewness** : 0.23
- **Kurtosis** : 0.30



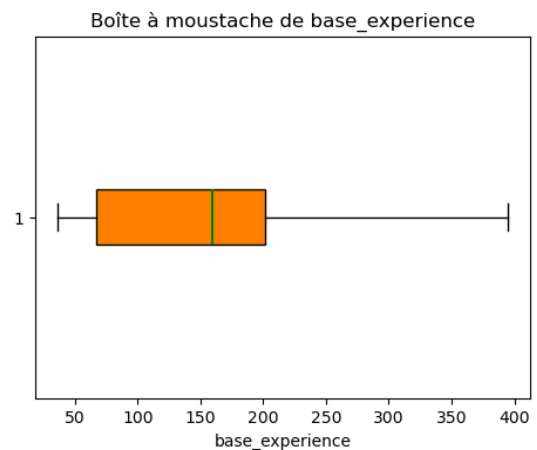
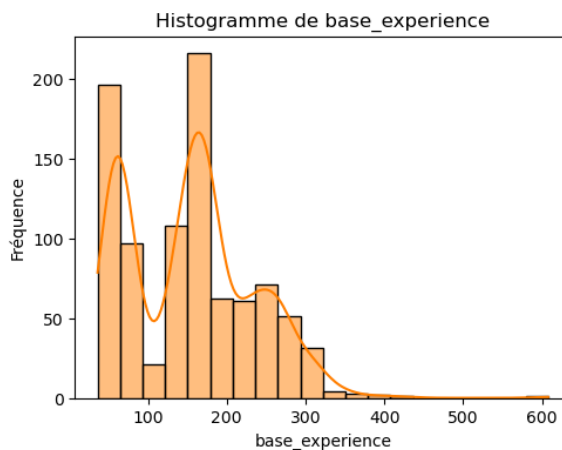
Analyse univariée : *catch_rate*

- **Count** : 1027
- **Mean** : 92.74
- **Std** : 75.88
- **Min** : 3.00
- **25%** : 45.00
- **50% (Median)** : 60.00
- **75%** : 127.00
- **Max** : 255.00
- **Skewness** : 0.95
- **Kurtosis** : -0.37



Analyse univariée : *base_experience*

- **Count** : 925
- **Mean** : 153.72
- **Std** : 79.28
- **Min** : 36.00
- **25%** : 67.00
- **50% (Median)** : 159.00
- **75%** : 201.00
- **Max** : 608.00
- **Skewness** : 0.53
- **Kurtosis** : 0.39



Il est important de noter que les boîtes à moustaches ont été tronquées et que les valeurs "outliers" ont été exclues de l'affichage, car certaines variables sont très éparpillées et auraient rendu l'affichage illisible

Analyse des variables et hypothèses :

Pour la plupart des variables, un Pokémon se distingue particulièrement et génère des valeurs absurdes (surtout dans masse et taille) : Ethernatos Etermax (pokémon le plus grand, plus lourd et avec le plus de statistiques de base pour sa version Infinimax), on ignorera la plupart des observations causées par ce pokémon, pour se concentrer sur un tout plutôt qu'une anomalie.

1. height

La variable height est très concentrée sur l'intervalle [0.6;1.6] et n'est pas du tout symétrique. La valeur de kurtosis et l'écart type indiquent tous les deux que les valeurs sont fortement condensées. La taille maximale observée est de 100m.

Considéré le manque d'hétérogénéité de la variable, il est peu probable que la taille ait une influence sur l'existence d'un profil type de Pokémon (pour les pokemons

Normaux). Il est cependant possible que les pokémons légendaires présentent une taille plus élevée

2. weight

Même remarque que pour height, la variable weight est plus dispersée, mais ne présente pas beaucoup plus de symétrie. la valeur kurtosis et l'écart-type étant très élevés, cela prouve que la densité est fortement concentrée autour de la moyenne. weight a aussi probablement peu d'impact dans le profilage des Pokémons normaux.

Hypothèse (1) : Comme les deux variables weight et height présentent des profils d'histogrammes similaires, il est logique de penser que plus un Pokémon est grand; plus il est susceptible d'être lourd (et inversement). La matrice de corrélation nous aidera sûrement à voir si les deux variables sont effectivement fortement corrélées ou non.

3. total_points

La variable total points se comporte de manière assez logique et est assez symétrique en plus d'être équilibrée . on remarque qu'elle pointe surtout autour des valeurs ~300 et ~500. Autour de ces valeurs, on remarque quand même une fréquence assez élevée d'autres Pokémons. les indices Skewness et Kurtosis sont très faibles, ce qui indique une bonne répartition des valeurs, malgré un écart type important. On observe une brutale chute de la fréquence à partir de 600.

Hypothèses :

- (2) Les Pokémons à plus de 600 points au total sont probablement corrélés à leur statut (normal, demi-légendaire, légendaire, mythique, etc.)
- (3) Les plus gros pics de fréquence sont dus aux Pokémons ayant deux évolutions, une donnée que nous n'avons pas dans ce dataset, mais qui serait intéressante à analyser et qui me semble le plus probable, comme la plupart des pokémons ont deux ou trois évolutions, les stats doivent plus ou moins finir en quinconce, car en général, les pokémons à trois evolutions sont plus puissants que ceux à 1 ou 2.

4. catch_rate

La variable catch_rate est une variable assez difficile à interpréter car elle dépend de la méthode de calcul du jeu utilisé, et il n'est pas précisé, que ce soit dans le sujet ou sur le dataset dans kaggle, de quel jeu est extrait le rate proposé ici. il sera donc nécessaire de faire une analyse rapide pour savoir comment interpréter la valeur du catch_rate d'un Pokémon. d'après bulbapedia (Poképédia anglais), le taux attribué à chaque pokemon est toujours situé entre 3 et 255, même si l'algorithme de calcul pour la capture est différent dans chaque jeu (voir [List of Pokémons by catch rate](#)). La variable est assez bien équilibrée (en regardant l'indice Kurtosis), et présente plusieurs gros pics à [~10, 45, 75, 90, ~120, 190, 255].

Hypothèses :

- (4) les valeurs de catch_rate, qui se situent entre 3 et 255, indiquent que plus le taux

est bas, plus il est difficile de capturer le Pokémon.

- (5) Plus un pokémon est difficile à capturer, plus il est puissant.

5. base_experience

Contrairement à ce qu'on pourrait penser au départ, la variable 'base_experience' ne représente pas le taux de base (au niveau 1) d'expérience à obtenir pour faire augmenter le Pokémon. Ce taux représente simplement la quantité d'expérience obtenue lorsque le pokemon (sauvage) est vaincu. Il existe bel et bien une variable qui définit la quantité à obtenir pour faire augmenter le niveau, ou à quelle vitesse notre Pokémon évolue, mais c'est plus une question de catégorie de "courbe", plutôt qu'une variable individuelle. la variable est bien équilibrée et assez symétrique (voir nos deux indices habituels), on note que chaque Pokémon rapporte au minimum 36 points d'expériences quand ils sont vaincus.

Pour plus de détails sur le fonctionnement de l'expérience : [Courbe d'expérience](#) (Poképédia) Pour faire simple, voici le diagramme représentant les différentes

courbes possibles (source : Poképédia) : 

Hypothèses :

- (6) Plus un pokémon est rapporte d'experience une fois vaincu, plus il est puissant.
- (7) Plus un pokémon rapporte d'experience une fois vaincu, plus il est difficile à capturer.

```
In [12]: combat_stats = ['hp', 'attack', 'defense', 'sp_attack', 'sp_defense', 'speed']

min_vals = dfPokemon[combat_stats].min().values
max_vals = dfPokemon[combat_stats].max().values

min_vals = np.append(min_vals, min_vals[0])
max_vals = np.append(max_vals, max_vals[0])

angles = np.linspace(0, 2 * np.pi, len(combat_stats), endpoint=False)
angles = np.append(angles, angles[0])

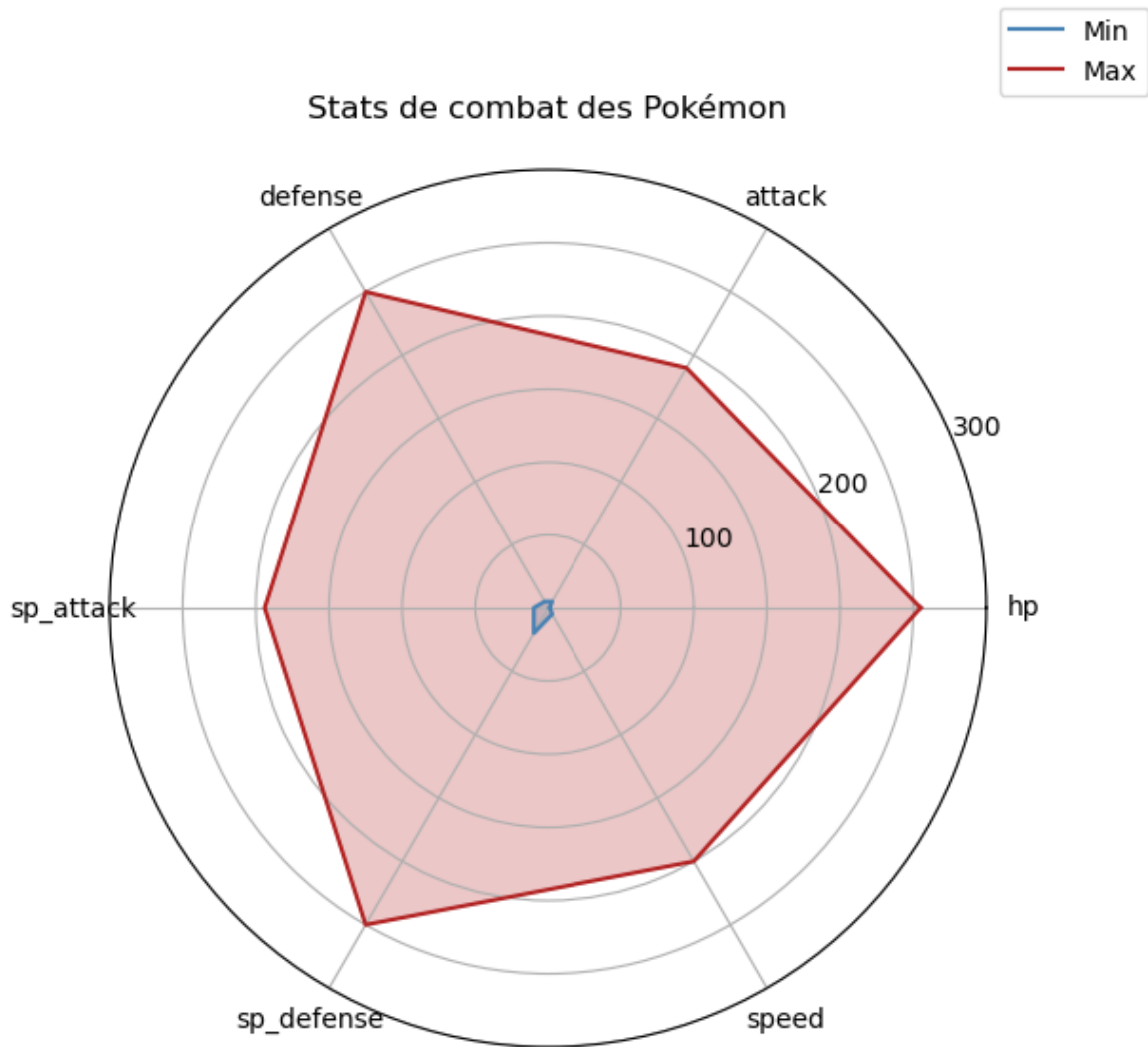
fig, ax = plt.subplots(figsize=(6, 6), subplot_kw=dict(polar=True))

ax.plot(angles, min_vals, label='Min', color='steelblue')
ax.fill(angles, min_vals, color='steelblue', alpha=0.25)

ax.plot(angles, max_vals, label='Max', color='firebrick')
ax.fill(angles, max_vals, color='firebrick', alpha=0.25)

ax.set_xticks(angles[:-1])
ax.set_xticklabels(combat_stats)
ax.set_yticks([50, 100, 150, 200, 250, 300])
ax.set_yticklabels(['', '100', '', '200', '', '300'])
ax.set_title("Stats de combat des Pokémon", pad=20)
ax.legend(loc='upper right', bbox_to_anchor=(1.2, 1.2))

plt.show()
```



On voit bien que la répartition des statistiques est très parsemée.

On va ajuster le minimum (on le multiplie x3) pour faire une analyse plus poussée :

```
In [14]: min_coeff = 3.0

min_vals_adjusted = min_vals * min_coeff

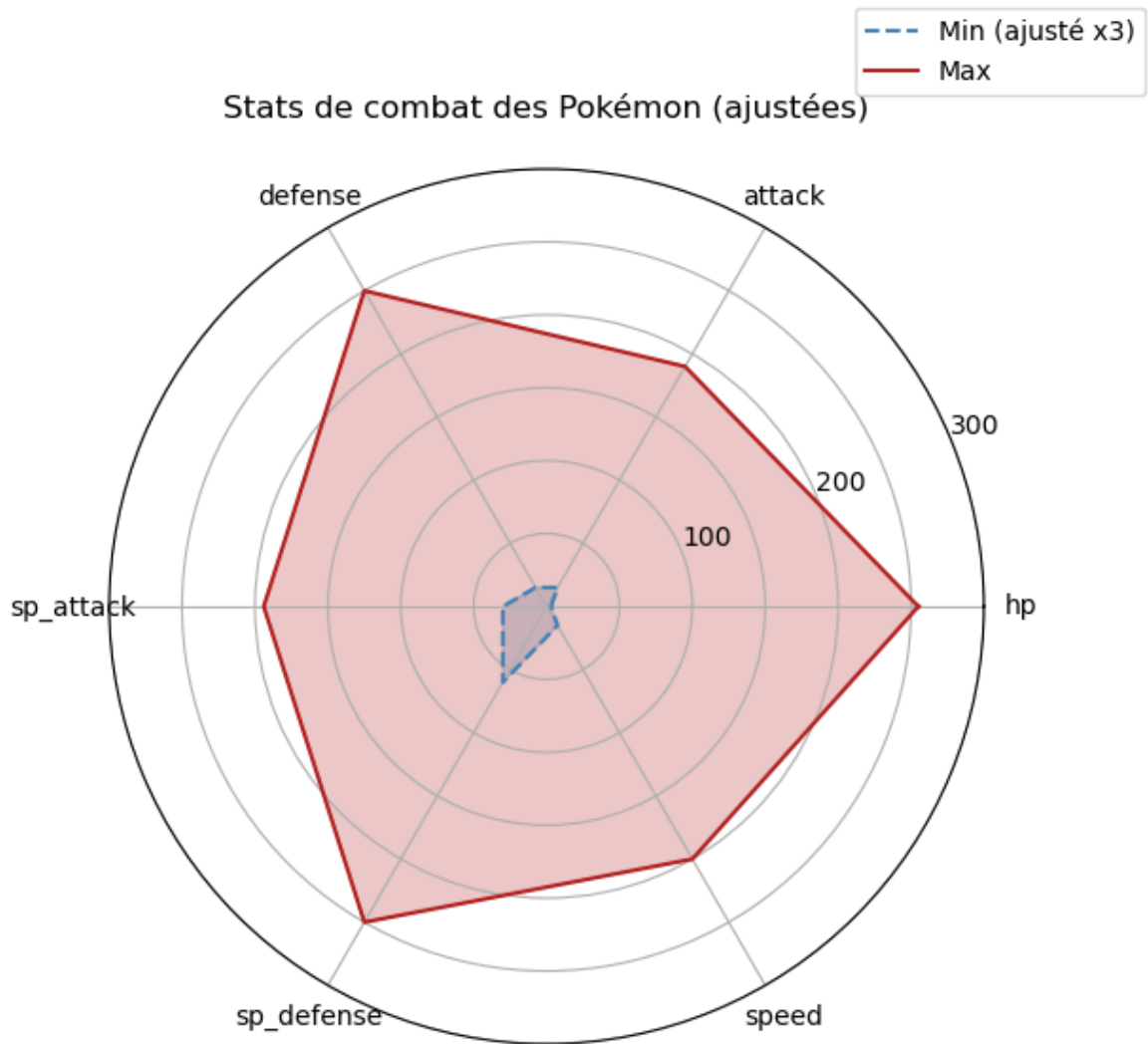
fig, ax = plt.subplots(figsize=(6, 6), subplot_kw=dict(polar=True))

ax.plot(angles, min_vals_adjusted, label='Min (ajusté x3)', color='steelblue', 1)
ax.fill(angles, min_vals_adjusted, color='steelblue', alpha=0.25)

ax.plot(angles, max_vals, label='Max', color='firebrick')
ax.fill(angles, max_vals, color='firebrick', alpha=0.25)

ax.set_xticks(angles[:-1])
ax.set_xticklabels(combat_stats)
ax.set_yticks([50, 100, 150, 200, 250, 300])
ax.set_yticklabels(['', '100', '', '200', '', '300'])
ax.set_title("Stats de combat des Pokémon (ajustées)", pad=20)
ax.legend(loc='upper right', bbox_to_anchor=(1.2, 1.2))

plt.show()
```



On remarque assez facilement que :

- hp est la variable avec l'écart min/max le plus élevé (et le minimum le plus bas)
- paradoxalement, hp est aussi la variable la plus élevée
- sp_def est la moins écartée et a le minimum le plus élevé
- attack a le maximum le plus bas

On ne remarque cependant pas de tendance particulière et que les pokémons sont très différents les uns des autres, ce qui consolide l'hypothèse d'une possible catégorisation ou l'existence de "profils types"

On trace un graphique type boxplot et violon sur ces stats pour faire apparaître la densité et la fréquence des valeurs :

```
In [16]: fig, axes = plt.subplots(1, 2, figsize=(18, 8))

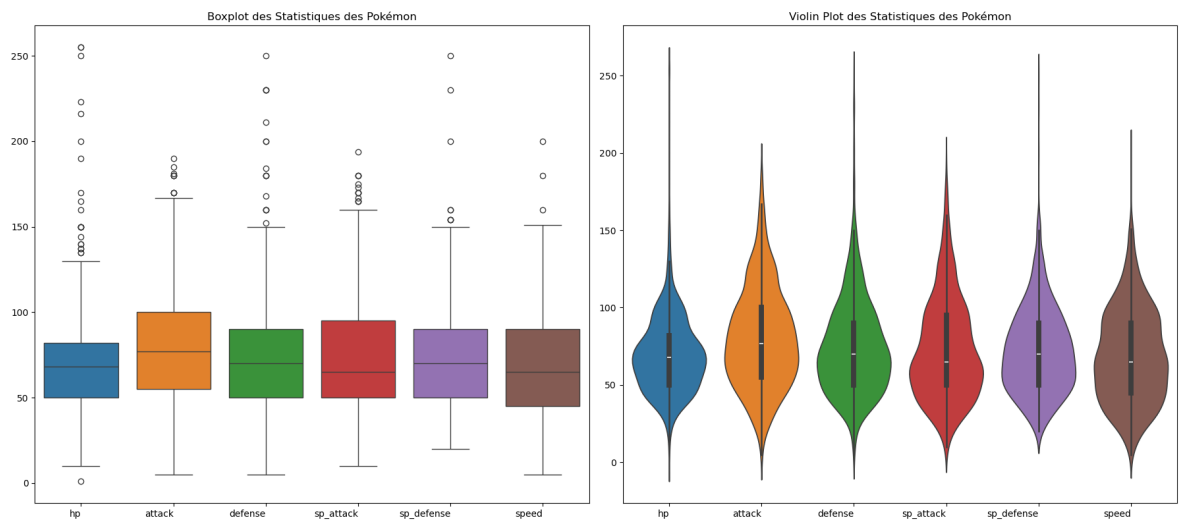
sns.boxplot(data=dfPokemon[combat_stats], ax=axes[0])
axes[0].set_title('Boxplot des Statistiques des Pokémon')
axes[0].set_xticks(range(len(combat_stats)))
axes[0].set_xticklabels(axes[0].get_xticklabels(), ha='right')

sns.violinplot(data=dfPokemon[combat_stats], ax=axes[1])
axes[1].set_title('Violin Plot des Statistiques des Pokémon')
axes[1].set_xticks(range(len(combat_stats)))
```



```
axes[1].set_xticklabels(axes[1].get_xticklabels(), ha='right')

plt.tight_layout()
plt.show()
```



On remarque que la moyenne la plus haute est celle de l'attaque, et sp_attack la plus faible paradoxalement

Les valeurs extrêmes sur la variable hp apparaissent d'autant plus sur la boîte à moustache.

Sur le diagramme en violon, on voit que speed est la variable la plus répartie et hétérogène, la densité étant fortement élevée tout le long du violon

Dans l'ensemble, toutes les variables semblent assez bien réparties, et sont équilibrées, une variable ne prend pas le dessus sur les autres, et on se retrouve avec 95% des valeurs entre 20 et 200

Analyse des variables qualitatives

On effectue désormais une analyse univariée des variables qualitatives

On commence par la répartition des types dans notre dataset :

```
In [19]: type_counts = dfPokemon['type_1'].value_counts() + dfPokemon['type_2'].value_cou
type_counts = type_counts.sort_values(ascending=True)

type_color_dict = {
    'Normal': '#A8A77A',
    'Fire': '#EE8130',
    'Water': '#6390F0',
    'Electric': '#F7D02C',
    'Grass': '#7AC74C',
    'Ice': '#96D9D6',
    'Fighting': '#C22E28',
    'Poison': '#A33EA1',
    'Ground': '#E2BF65',
    'Flying': '#A98FF3',
    'Psychic': '#F95587',
```

```

    'Bug': '#A6B91A',
    'Rock': '#B6A136',
    'Ghost': '#735797',
    'Dragon': '#6F35FC',
    'Dark': '#705746',
    'Steel': '#B7B7CE',
    'Fairy': '#D685AD'
}

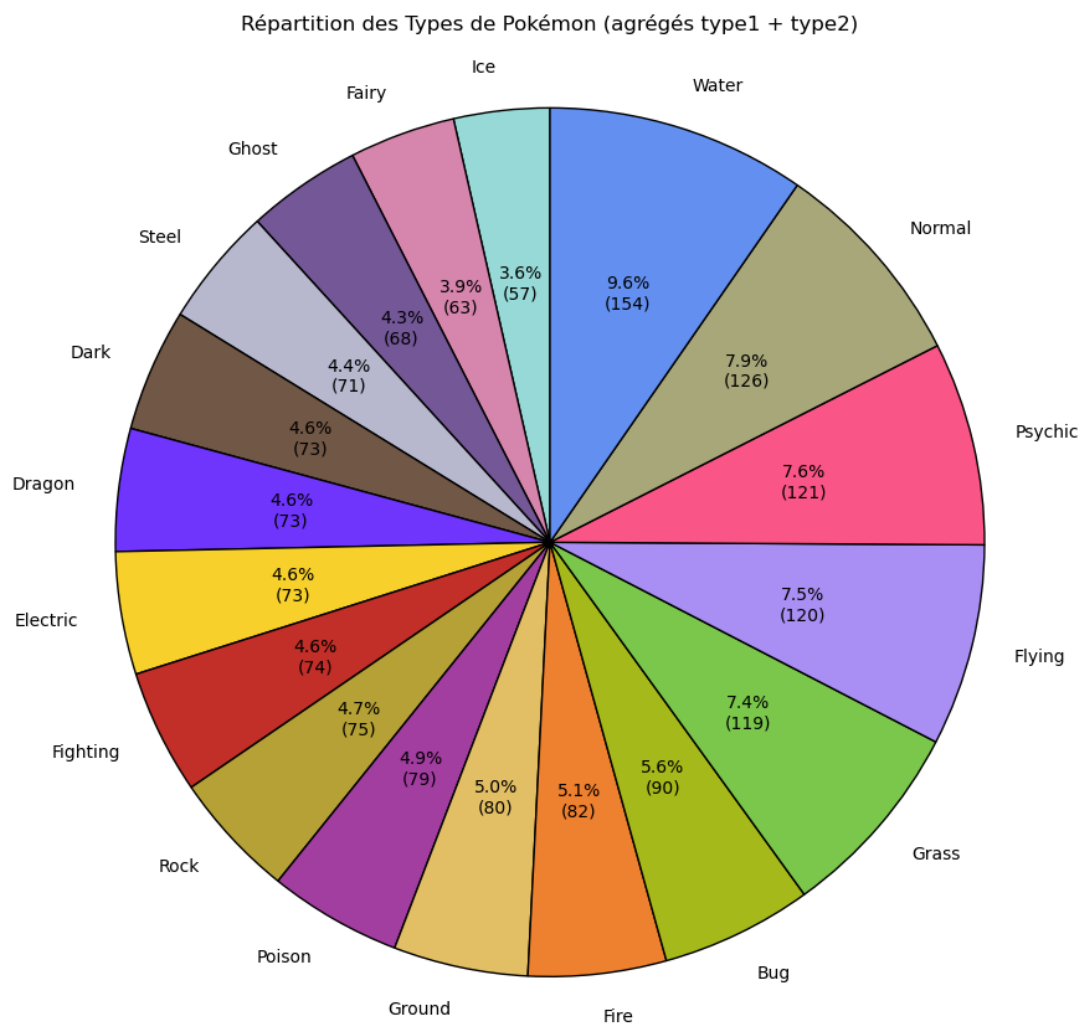
colors = [type_color_dict[type_] for type_ in type_counts.index]

def custom_autopct(pct, allvals):
    total = sum(allvals)
    val = int(round(pct * total / 100.0))
    return f'{pct:.1f}%\n({val:d})'

plt.figure(figsize=(12, 10))
plt.pie(type_counts, labels=type_counts.index, autopct= lambda pct: custom_autopct(
plt.title('Répartition des Types de Pokémon (agrégés type1 + type2)', pad=20)
plt.axis('equal')
plt.show()

print(f"Total : {type_counts.sum()}")

```



Total : 1598

Observations :

- Water est la variable la plus représentée avec 154 Pokémon ayant au moins un des

deux de leur type étant Eau.

- Ice est la moins représentée avec seulement 57 Pokémons, c'est assez surprenant.
- On sait que le total représenté ici est de 1598 types. on sait donc que parmi nos 1045 lignes -> $1598 - 1045 = 553$ Pokémons ayant un double type ($553/1045 = 52.9\%$ des pokémons ont un double type !)
- Il est en soit logique que le type "normal" se hisse en deuxième place, car la plupart des pokémons de type "vol" des anciennes générations ont le double type "normal"

Les types semblent être aussi bien équilibrés que les statistiques présentées au dessus.

Hypothèses :

- (8) Il existe un type de Pokémon présentant des statistiques plus élevées que les autres.
- (9) Il existe un type de Pokémon plus difficile à attraper que les autres.

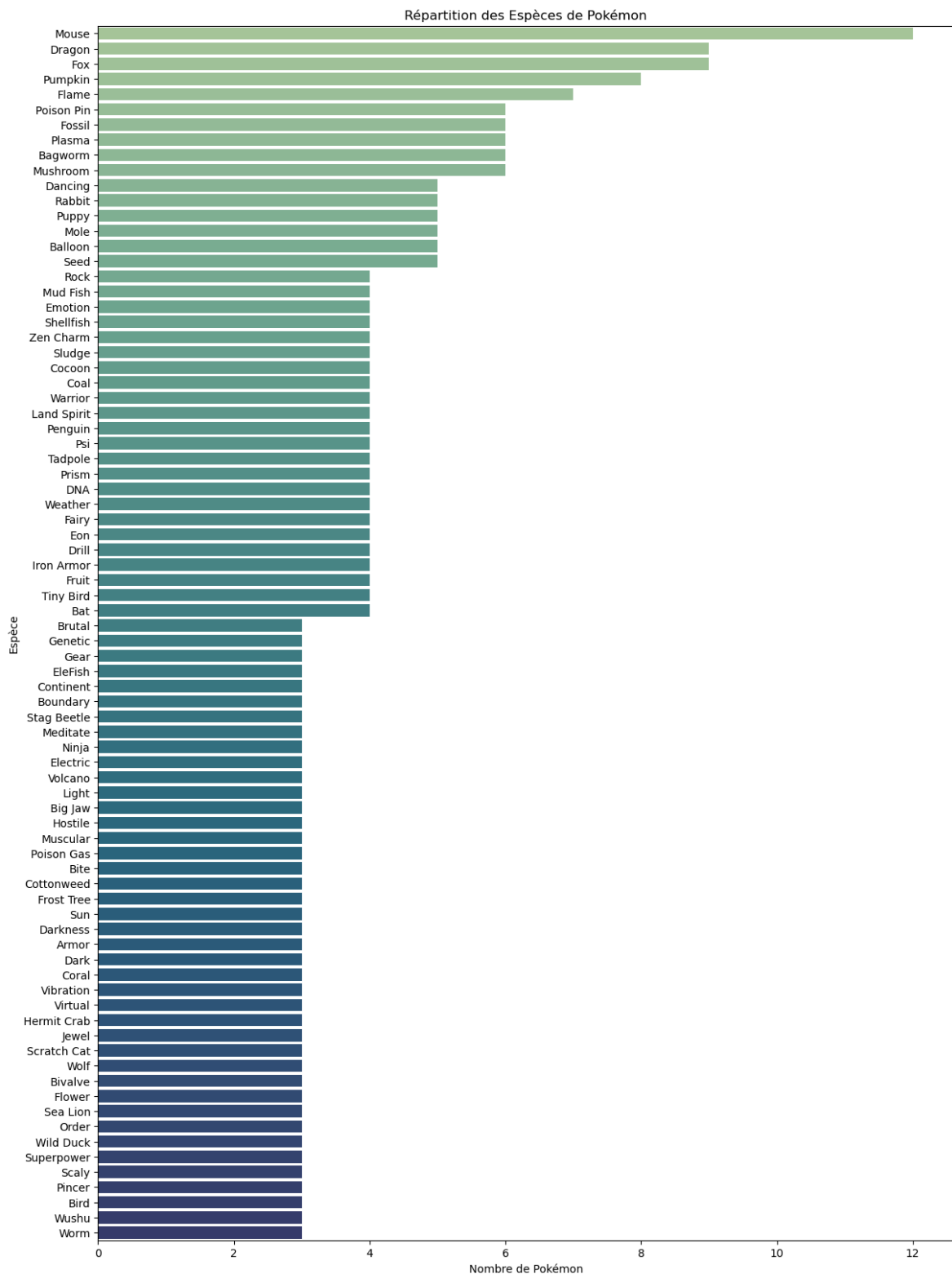
Passons à la répartition des espèces :

On a essayé d'afficher nos espèces sous forme de barplot, mais il semblerait que la variable espèce contienne trop de valeurs différentes, ce qui limite fortement son intérêt dans notre usage ici. On va donc simplement seulement récupérer les espèces avec strictement plus de 2 représentants par type. On les affichera dans un barplot.

```
In [21]: species_counts = dfPokemon['species'].value_counts()

displayed_species_counts = species_counts[species_counts.values > 2]
displayed_species_counts

plt.figure(figsize=(14, 20))
sns.barplot(y=displayed_species_counts.index, x=displayed_species_counts.values,
plt.title('Répartition des Espèces de Pokémon')
plt.xlabel('Nombre de Pokémon')
plt.ylabel('Espèce')
plt.show()
```



On observe que les 5 espèces les plus représentées sont : Mouse (12 occurrences), Dragon (9 occurrences), Fox (9 occurrences), Pumpkin (8 occurrences) et Flame (7 occurrences)

Il apparait clairement que les espèces de pokémons sont très diversifiées, car la plupart des espèces ne possèdent qu'un seul ou deux représentant(s)

```
In [23]: print(f"Seulement {round(displayed_species_counts.value_counts().sum() / species
```

Seulement 12.27% des espèces ont été représentées dans ce graphique, ce qui prouve que les Pokémon sont très diversifiés

Hypothèse : (10) Les espèces les plus représentées fondent-elles un profil type en

statistiques ? en masse/taille ? en types ? en taux de capture ?

Continuons en étudiant les statuts des pokémon (Normal, Sous-légendaire, légendaire, mythique) :

```
In [25]: status_counts = dfPokemon['status'].value_counts()

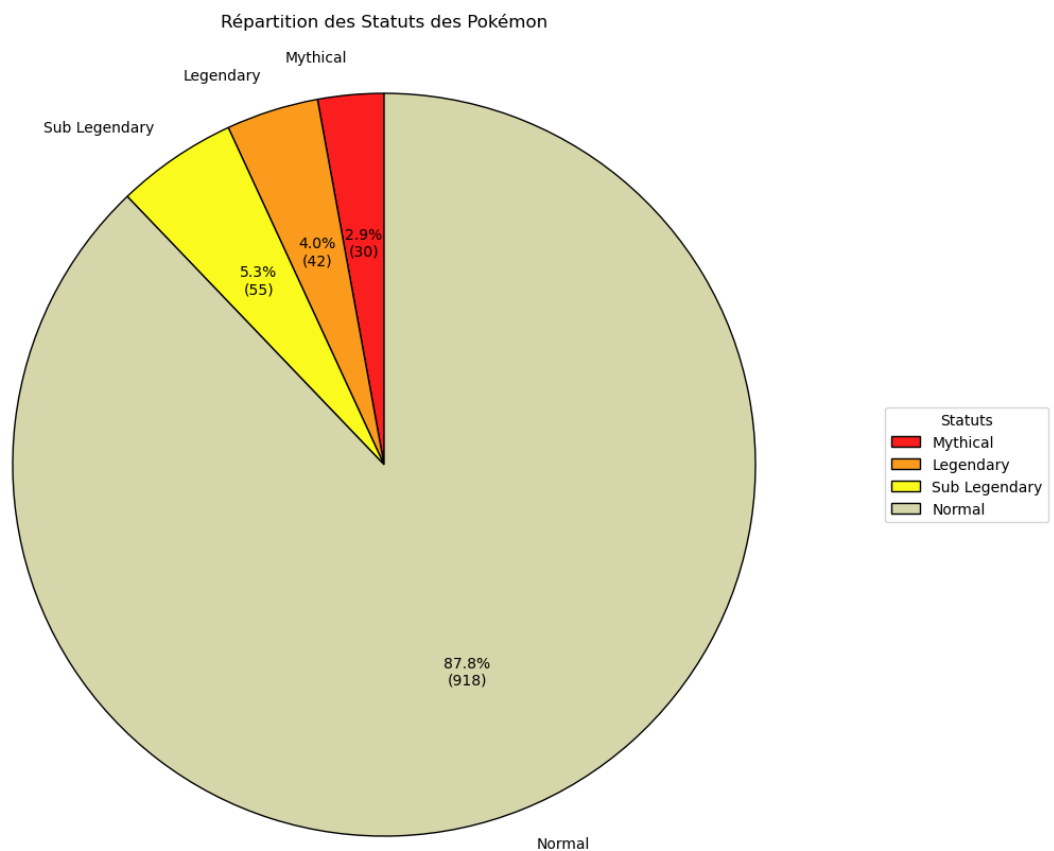
status_counts = status_counts.sort_values(ascending=True)

status_colors = ['#fa1e1e', '#fa9b1e', '#fafa1e', '#d6d6ab']

def custom_autopct(pct, allvals):
    total = sum(allvals)
    val = int(round(pct * total / 100.0))
    return f'{pct:.1f}%\n({val:d})'

plt.figure(figsize=(12, 10))
wedges, texts, autotexts = plt.pie(status_counts, labels=status_counts.index, au
plt.title('Répartition des Statuts des Pokémon', pad=20)
plt.axis('equal')

plt.legend(wedges, status_counts.index, title="Statuts", loc="center left", bbox
plt.show()
```



Il existe en tout 4 statuts de pokémons : Normal, Semi-légendaire, Légendaire, Mythique.

On observe que 87.8% des pokémons sont normaux, ce qui est un score en réalité assez bas quand on y réfléchit une deuxième fois.

En termes de statistiques de base, les Pokémon de statut 'Légendaire' ont en moyenne des statistiques de base plus élevées que les Pokémon de statut 'Normal'. On explorera cette question plus tard dans le document...

Hypothèses/Questions :

- (11) Quelle est réellement la différence de statistiques entre les pokémons légendaires et non légendaires ?
- (12) Y-a t'il un type plus représenté parmi les pokémons non normaux ?

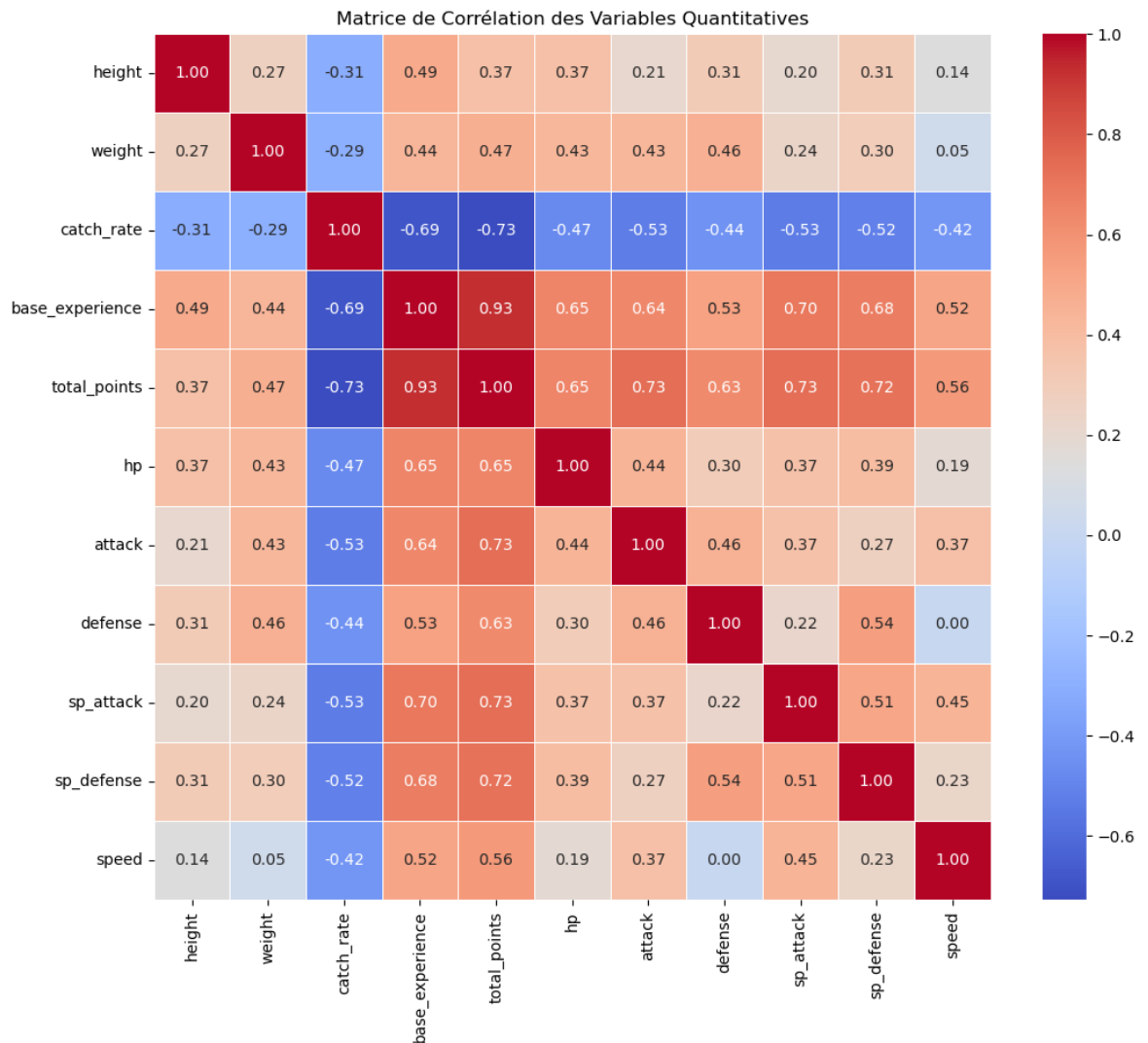
Analyse Bivariée/Multivariée

On essaiera de répondre à toutes les questions/hypothèses posées dans l'analyse univariée, **à noter que les questions ne seront pas traitées dans l'ordre, pour ajouter du rythme/des catégories au document**

Pour continuer notre analyse, nous allons passer à l'analyse bivariée/multivariée, en commençant par la matrice de corrélation, pour montrer à quel point les variables quantitatives sont corrélées les unes aux autres :

```
In [28]: quantitative_columns = ['height', 'weight', 'catch_rate', 'base_experience', 'total_experience']
correlation_matrix = dfPokemon[quantitative_columns].corr()

plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f", linewidths=1)
plt.title('Matrice de Corrélation des Variables Quantitatives')
plt.show()
```



Beaucoup d'observations intéressantes dans ce graphique :

Commençons par le plus évident, une forte corrélation entre le total de points et les différentes statistiques, rien de surprenant.

Notons quand même un taux à 0.54 entre def et sp_def.

les autres statistiques ne semblent pas avoir un taux trop corrélé les unes aux autres.

speed a l'air d'être la variable la plus indépendante de la liste.

Réponse à l'hypothèse (1) :

Avec un coefficient de 0.27, le taux ne suffit pas à prouver une corrélation entre les deux variables.

On apporte un test d'indépendance dans la cellule suivante pour affirmer ou non cette observation :

```
In [30]: correlation, p_value = pearsonr(dfPokemon['height'], dfPokemon['weight'])
print(f"Coefficient de corrélation de Pearson : {correlation:.2f}")
```

```
print(f"P-value : {p_value:.4f}")

if p_value < 0.05:
    print("Il existe une corrélation significative entre 'height' et 'weight'.")
else:
    print("Il n'existe pas de corrélation significative entre 'height' et 'weigh
```

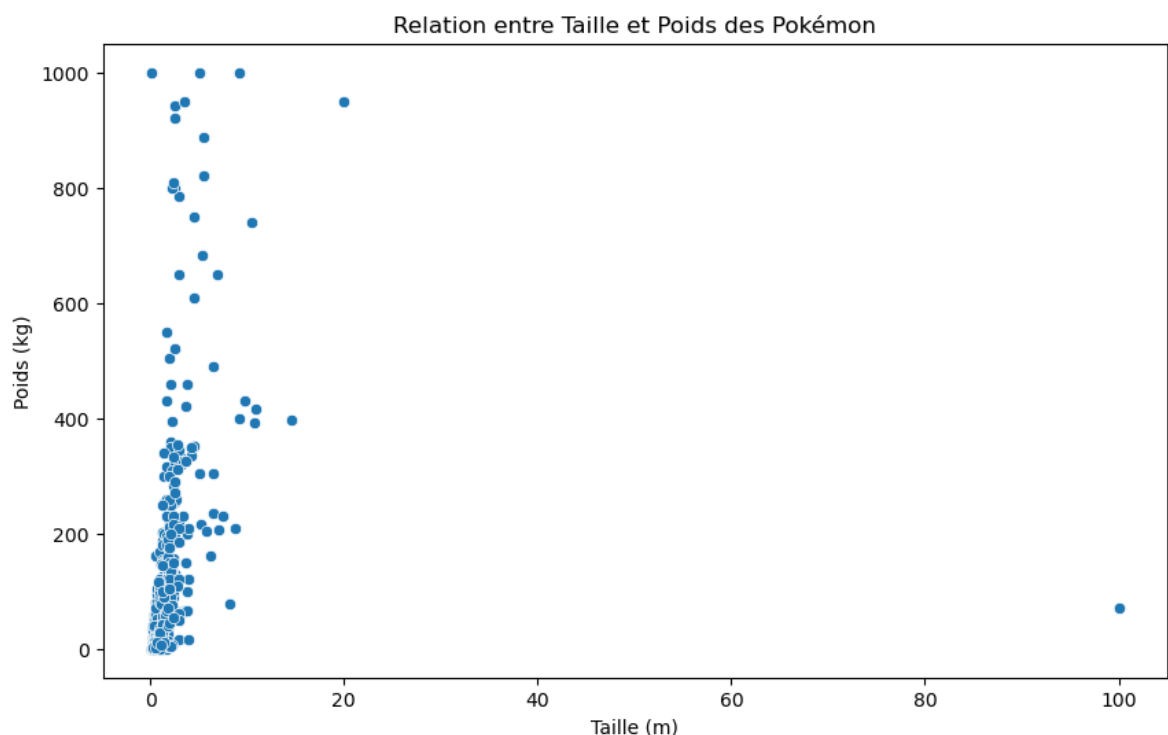
Coefficient de corrélation de Pearson : 0.27

P-value : 0.0000

Il existe une corrélation significative entre 'height' et 'weight'.

On utilise un scatterplot pour afficher visuellement la corrélation entre les deux variables :

```
In [32]: plt.figure(figsize=(10, 6))
sns.scatterplot(x='height', y='weight', data=dfPokemon)
plt.title('Relation entre Taille et Poids des Pokémon')
plt.xlabel('Taille (m)')
plt.ylabel('Poids (kg)')
plt.show()
```



Il semblerait que d'après notre test de corrélation (pearson), il existe une corrélation significative entre height et weight.

Cependant, même si la corrélation est significative, elle reste assez faible dans les faits.

Ce résultat est sûrement dû au fait que nous analysons plusieurs centaines de Pokémon à la fois.

On peut donc en conclure que les deux variables sont liées, mais de manière forte ou linéaire.

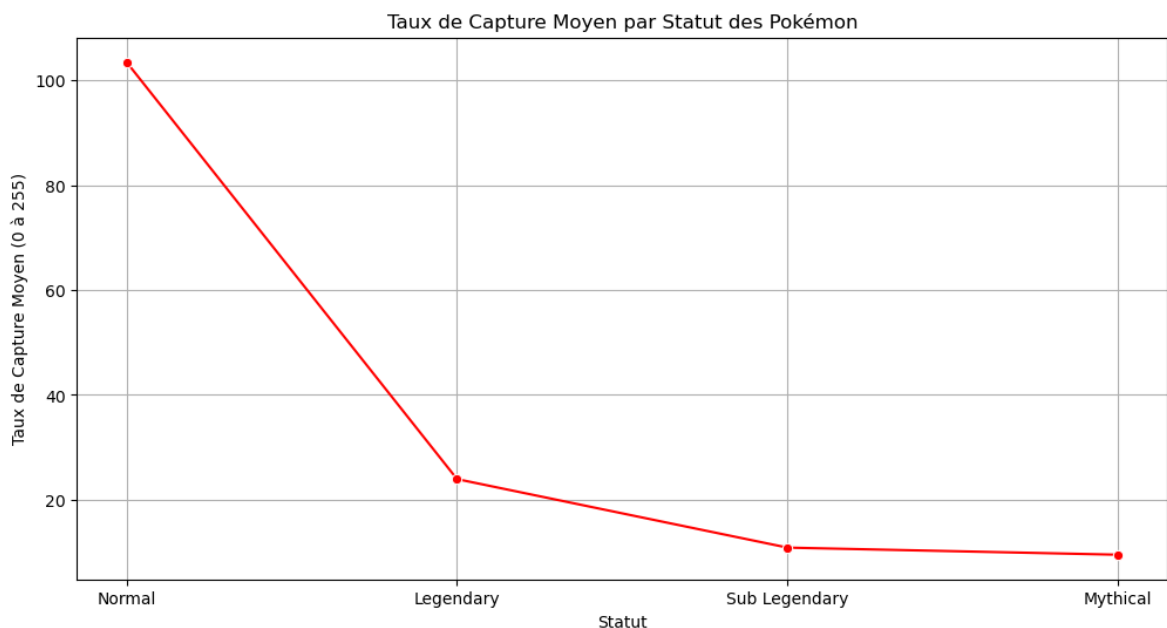
Nous allons désormais essayer de répondre à la question (4), ce qui nous permettra de répondre aux hypothèses (5) et (6).

En toute logique, les pokémons ayant le statut "**normal**" sont plus faciles à attraper que les pokémons ayant un autre statut.

On peut donc tracer un lineplot pour faire apparaître les taux de capture moyens de chaque groupe.

```
In [34]: catch_rate_by_status = dfPokemon.groupby('status')['catch_rate'].mean().reset_index()
catch_rate_by_status = catch_rate_by_status.sort_values(by='catch_rate', ascending=False)

plt.figure(figsize=(12, 6))
sns.lineplot(x='status', y='catch_rate', data=catch_rate_by_status, marker='o',
plt.title('Taux de Capture Moyen par Statut des Pokémon')
plt.xlabel('Statut')
plt.ylabel('Taux de Capture Moyen (0 à 255)')
plt.grid(True)
plt.show()
```



Réponse à la question (4) :

Le graphique montre clairement que plus un pokémon est rare (de son statut), plus il est difficile à capturer.

Une observation hors de notre hypothèse est que le taux de capture des sous-légendaires est plus bas que celui des légendaires, une explication possible serait que beaucoup de légendaires sont inclus à l'histoire principale des jeux, et ces pokémons ont volontairement un taux de capture plus élevé que les autres (dans leur version histoire principale)

Continuons l'analyse de notre matrice de corrélation :

Réponse à l'hypothèse (7) :

Il semble y avoir une corrélation très forte entre le taux de capture et l'expérience de base obtenue lors de la victoire contre un pokémon sauvage. la matrice de corrélation indique que plus le Pokémon rapporte de l'expérience, plus il est difficile à capturer (comme on sait que plus l'indice est bas, plus le pokémon est difficile à capturer).

Comme pour la masse et la taille, on va faire un test d'indépendance pour voir à quel point les deux variables sont liées :

```
In [37]: df_baseexp_catchrt = dfPokemon.dropna(subset=['base_experience', 'catch_rate']) #
correlation, p_value = pearsonr(df_baseexp_catchrt['base_experience'], df_baseexp_

print(f"Coefficient de corrélation de Pearson : {correlation:.2f}")
print(f"P-valeur : {p_value:.4f}")

if p_value < 0.05:
    print("Il existe une corrélation significative entre 'base_experience' et 'c
else:
    print("Il n'existe pas de corrélation significative entre 'base_experience' ,
```

Coefficient de corrélation de Pearson : -0.69

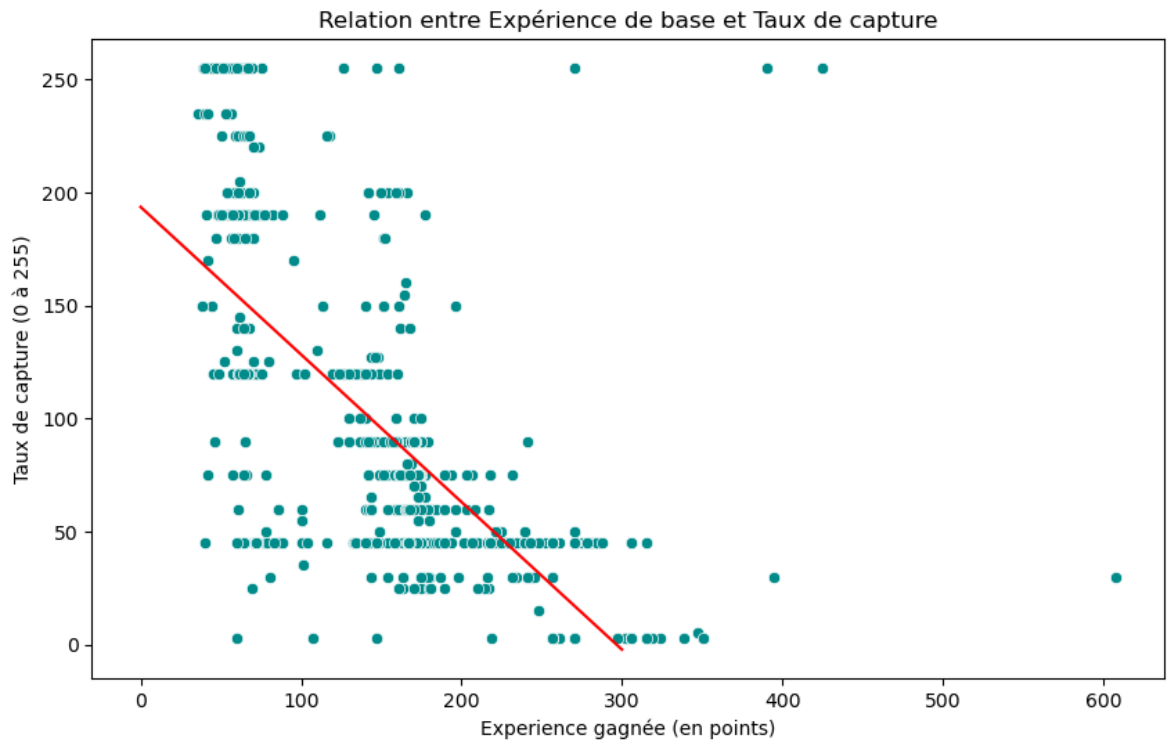
P-valeur : 0.0000

Il existe une corrélation significative entre 'base_experience' et 'catch_rate'.

On affiche notre résultat dans un scatterplot, en calculant en plus la régression :

```
In [39]: linreg = linregress(df_baseexp_catchrt['base_experience'],df_baseexp_catchrt['catc
linregmin = linreg.slope * 0 + linreg.intercept
linregmax = linreg.slope * 300 + linreg.intercept
dfLinReg = pd.DataFrame(data={'base_experience':[0,300], 'catch_rate':[linregmin

plt.figure(figsize=(10, 6))
sns.scatterplot(x='base_experience', y='catch_rate', data=dfPokemon, color='dark
plt.plot(dfLinReg['base_experience'],dfLinReg['catch_rate'],color='red')
plt.title('Relation entre Expérience de base et Taux de capture')
plt.xlabel('Expérience gagnée (en points)')
plt.ylabel('Taux de capture (0 à 255)')
plt.show()
```



On s'aperçoit très facilement que le taux de capture baisse drastiquement au fur et à mesure que les points d'expérience augmentent. La ligne de régression n'est pas tout à fait exacte, mais montre bien que le le taux baisse fortement.

Cette observation confirme donc l'hypothèse (7).

Réponse à l'hypothèse (5) :

Toujours sur la matrice de corrélation, on observe un taux assez élevé entre la puissance globale et le taux de capture des Pokémon.

On applique donc la même méthode que précédemment, comme on a une variable quantitative :

```
In [42]: df_total_catchrt = dfPokemon.dropna(subset=['catch_rate', 'total_points']) # on s
correlation, p_value = pearsonr(df_total_catchrt['total_points'], df_total_catchrt['catch_rate'])
print(f"Coefficient de corrélation de Pearson : {correlation:.2f}")
print(f"P-valeur : {p_value:.4f}")

if p_value < 0.05:
    print("Il existe une corrélation significative entre 'total_points' et 'catch_rate'")
else:
    print("Il n'existe pas de corrélation significative entre 'total_points' et 'catch_rate'")
```

Coefficient de corrélation de Pearson : -0.73

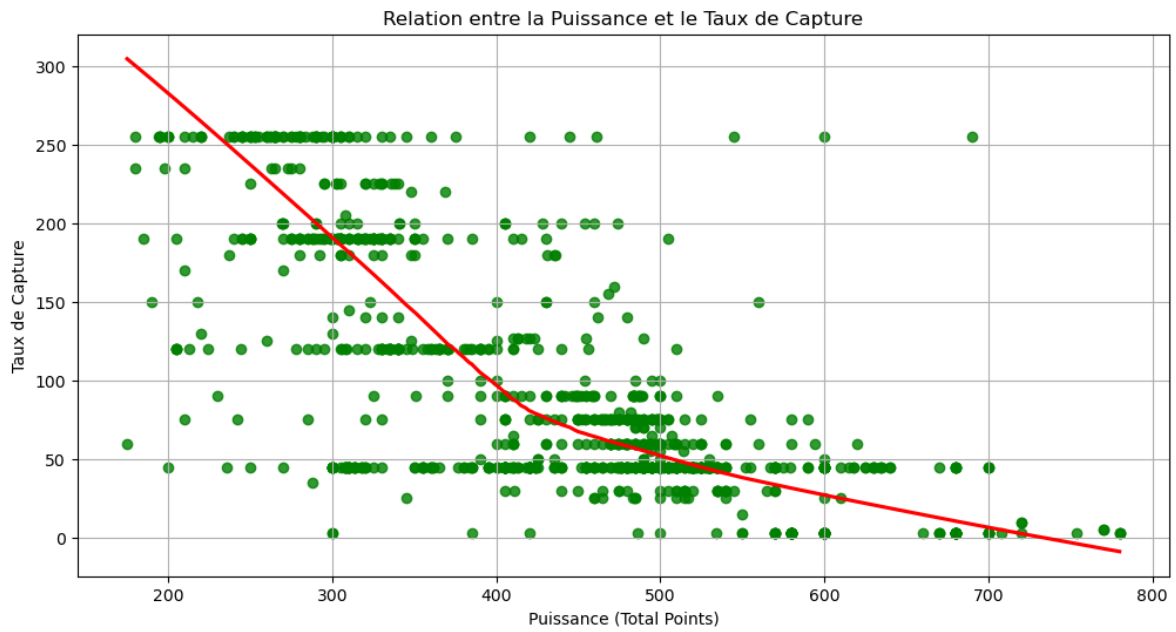
P-valeur : 0.0000

Il existe une corrélation significative entre 'total_points' et 'catch_rate'.

Comme pour les autres variables, il y a une corrélation significative entre les deux

On trace un regplot pour suivre l'évolution du taux de capture par rapport à la puissance :

```
In [44]: plt.figure(figsize=(12, 6))
sns.regplot(x='total_points', y='catch_rate', data=dfPokemon, line_kws={'color':
plt.title('Relation entre la Puissance et le Taux de Capture')
plt.xlabel('Puissance (Total Points)')
plt.ylabel('Taux de Capture')
plt.grid(True)
plt.show()
```



Comme pour l'hypothèse (7), on remarque que le taux de capture baisse fortement quand la puissance augmente.

Ce graphique confirme notre hypothèse (5).

Réponse à l'hypothèse (6) :

Pour terminer avec les analyses uniquement portées sur les variables quantitatives, et en toute logique, on observe la très forte corrélation entre `base_experience` et `total_points` :

```
In [47]: df_total_basexp = dfPokemon.dropna(subset=['base_experience', 'total_points']) #
correlation, p_value = pearsonr(df_total_basexp['total_points'], df_total_basexp
print(f"Coefficient de corrélation de Pearson : {correlation:.2f}")
print(f"P-valeur : {p_value:.4f}")

if p_value < 0.05:
    print("Il existe une corrélation significative entre 'base_experience' et 't
else:
    print("Il n'existe pas de corrélation significative entre 'base_experience'
```

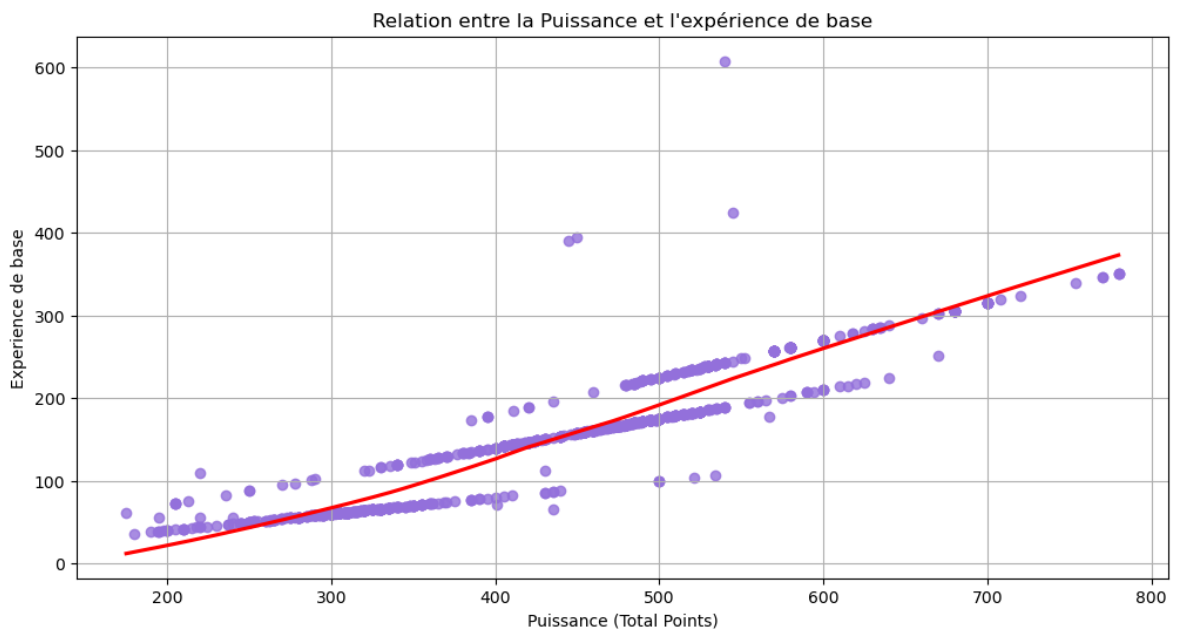
Coefficient de corrélation de Pearson : 0.93

P-valeur : 0.0000

Il existe une corrélation significative entre 'base_experience' et 'total_points'.

Même méthode qu'avant, on prend un regplot :

```
In [49]: plt.figure(figsize=(12, 6))
sns.regplot(x='total_points', y='base_experience', data=dfPokemon, line_kws={'co
plt.title('Relation entre la Puissance et l\'expérience de base')
plt.xlabel('Puissance (Total Points)')
plt.ylabel('Experience de base')
plt.grid(True)
plt.show()
```



On voit très clairement les trois lignes se former sur le graphique, les deux variables sont vraiment fortement liées.

L'hypothèse (6) est donc confirmée.

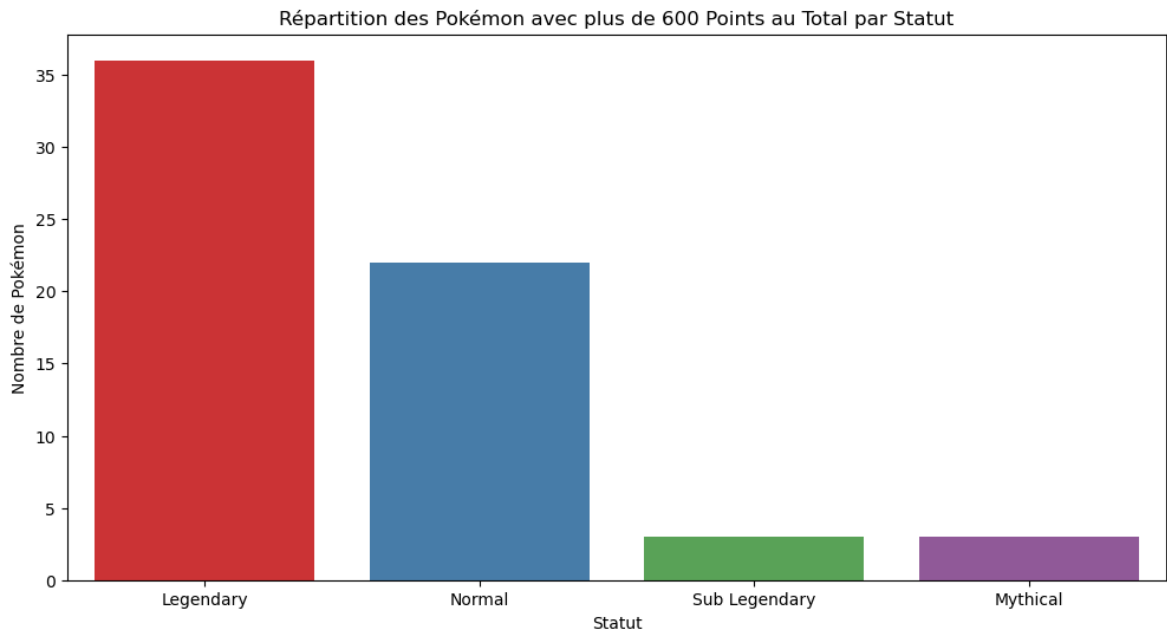
Passons maintenant à l'hypothèse (2):

On va simplement regarder le nombre de Pokémon dans chaque catégorie en filtrant tout ceux qui ont plus de 600 points dans un barplot :

```
In [51]: high_power_pokemon = dfPokemon[dfPokemon['total_points'] > 600]

status_counts = high_power_pokemon['status'].value_counts().reset_index()
status_counts.columns = ['status', 'count']

plt.figure(figsize=(12, 6))
sns.barplot(x='status', y='count', data=status_counts, hue='status', palette='Se
plt.title('Répartition des Pokémon avec plus de 600 Points au Total par Statut')
plt.xlabel('Statut')
plt.ylabel('Nombre de Pokémon')
plt.show()
```



On observe que la plupart des Pokémon au dessus de 600 points totaux sont en effet légendaires, mais il a quand même beaucoup de Pokémon normaux au dessus de 600 points.

On peut donc en tirer la conclusion suivante : Les Pokémon à plus de 600 points totaux sont moyennement liés à leur status, il y a certes plus de pokémons non normaux dans la liste, mais les pokémons normaux représentent quand même une partie non négligeable de la liste.

L'hypothèse (2) est donc rejetée.

Réponse à l'hypothèse (8) :

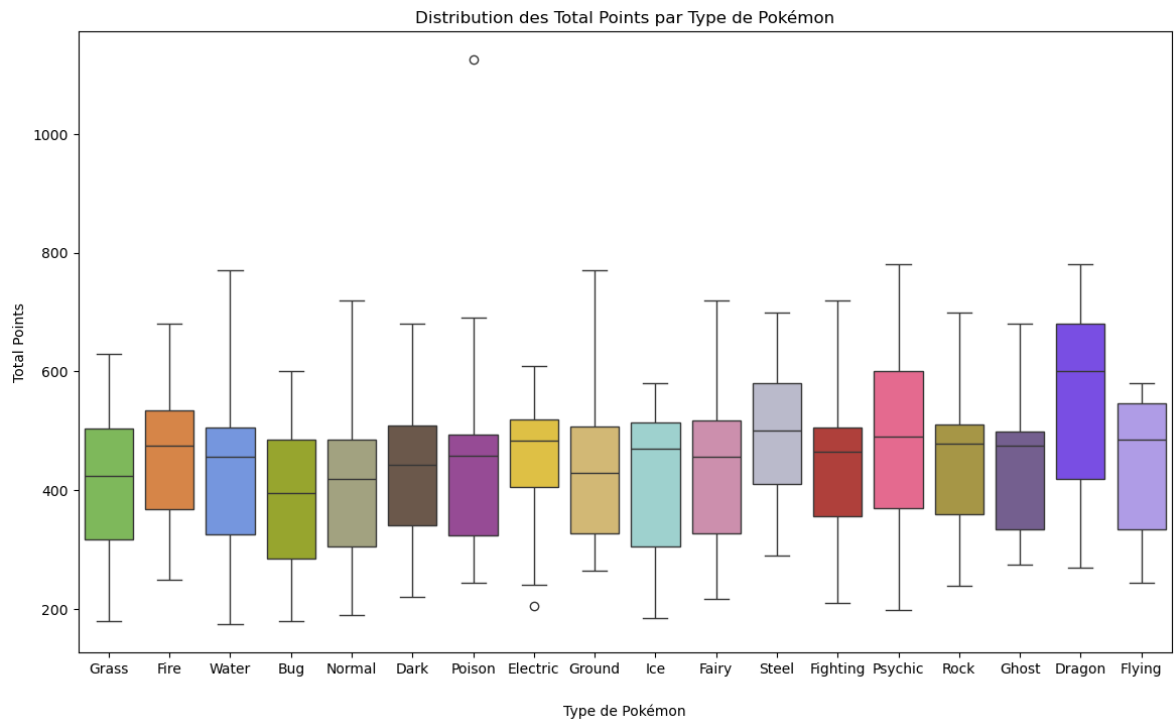
On trace un boxplot (type de graphique qui semble le plus évident pour ce genre de comparaisons) pour montrer les différences de statistiques entre tous les types :

J'ai essayé de faire un grouped bar plot, avec les minimum est maximums de chaque type, mais ça ne rendait pas aussi bien que le boxplot.

```
In [54]: colors = [type_color_dict[type_] for type_ in dfPokemon['type_1'].unique()]

plt.figure(figsize=(14, 8))
sns.boxplot(x='type_1', y='total_points', data=dfPokemon, hue='type_1', palette=
plt.title('Distribution des Total Points par Type de Pokémon')
plt.xlabel('Type de Pokémon', labelpad=20)
plt.ylabel('Total Points')

plt.show()
```



On remarque que les pokémons de type dragon présentent des statistiques beaucoup plus élevées que la moyenne.

Le type glace, déjà sous représenté, n'est pas très puissant avec l'un des minimum les plus bas.

Le type eau est le plus écarté.

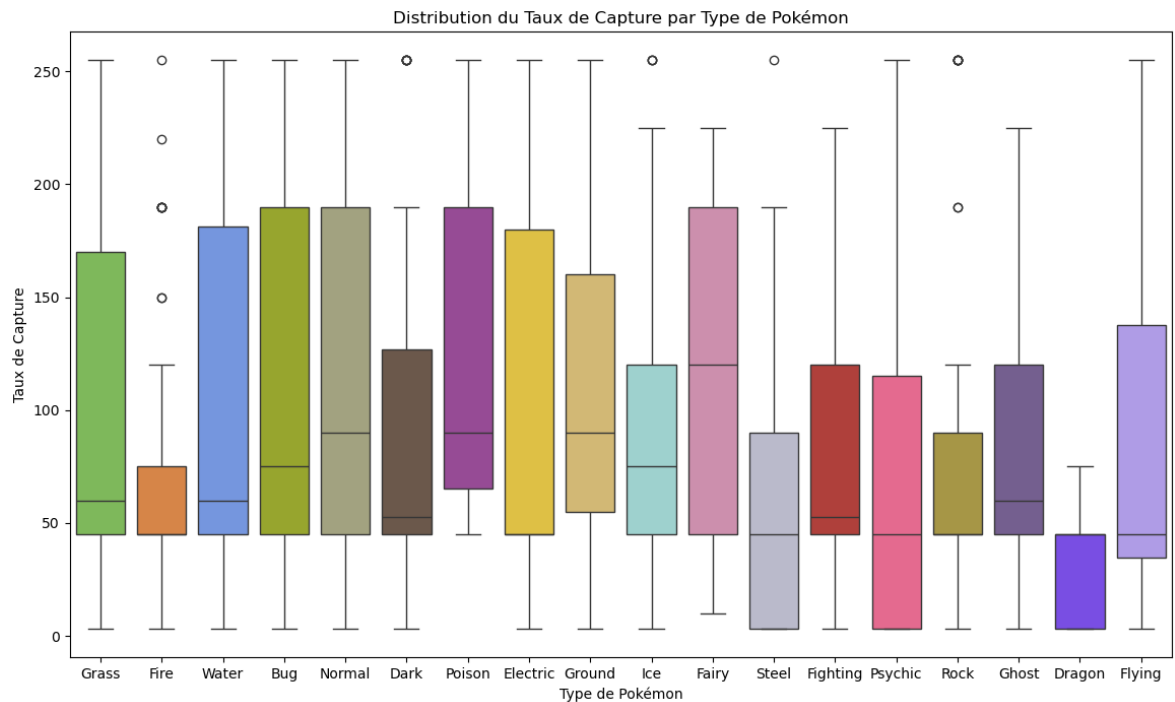
Les types qui se démarquent surtout sont : Feu, Eau, Electrique, Sol, Psy, Acier, Dragon et Vol.

Cela confirme notre hypothèse (8), en nous apportant la réponse suivante : Le type dragon semble avoir les pokémons avec les plus hautes statistiques.

Réponse à l'hypothèse (9) :

À l'image de la réponse précédente, comme on suit la même colonne, on cherche à définir les taux de capture les plus extrêmes par type. Nous allons donc utiliser la même méthode, en analysant un boxplot :

```
In [57]: plt.figure(figsize=(14, 8))
sns.boxplot(x='type_1', y='catch_rate', data=dfPokemon, hue='type_1', palette=co
plt.title('Distribution du Taux de Capture par Type de Pokémon')
plt.xlabel('Type de Pokémon')
plt.ylabel('Taux de Capture')
plt.show()
```



On remarque très facilement que les types les plus difficiles à attraper sont :

- Feu
- Acier
- Psy
- Roche
- Dragon (encore une fois)

Ce résultat est en réalité assez logique, comme on a une corrélation assez élevée entre la puissance et le taux de capture, les types les plus puissants sont aussi les plus durs à attraper.

Réponse : Oui il existe un type plus difficile à attraper, et celui dépend beaucoup de sa puissance, les type Acier et dragon semblent les plus durs à attraper

Réponse à la question (10) :

Pour la question (10), comme elle regroupe beaucoup de questions à la fois, on tracera d'abord tous nos graphiques et on en tirera une conclusion.

On limitera aussi le nombre d'espèces représentées à **20**, comme les Pokémon sont très diversifiés.

Voici nos graphiques :

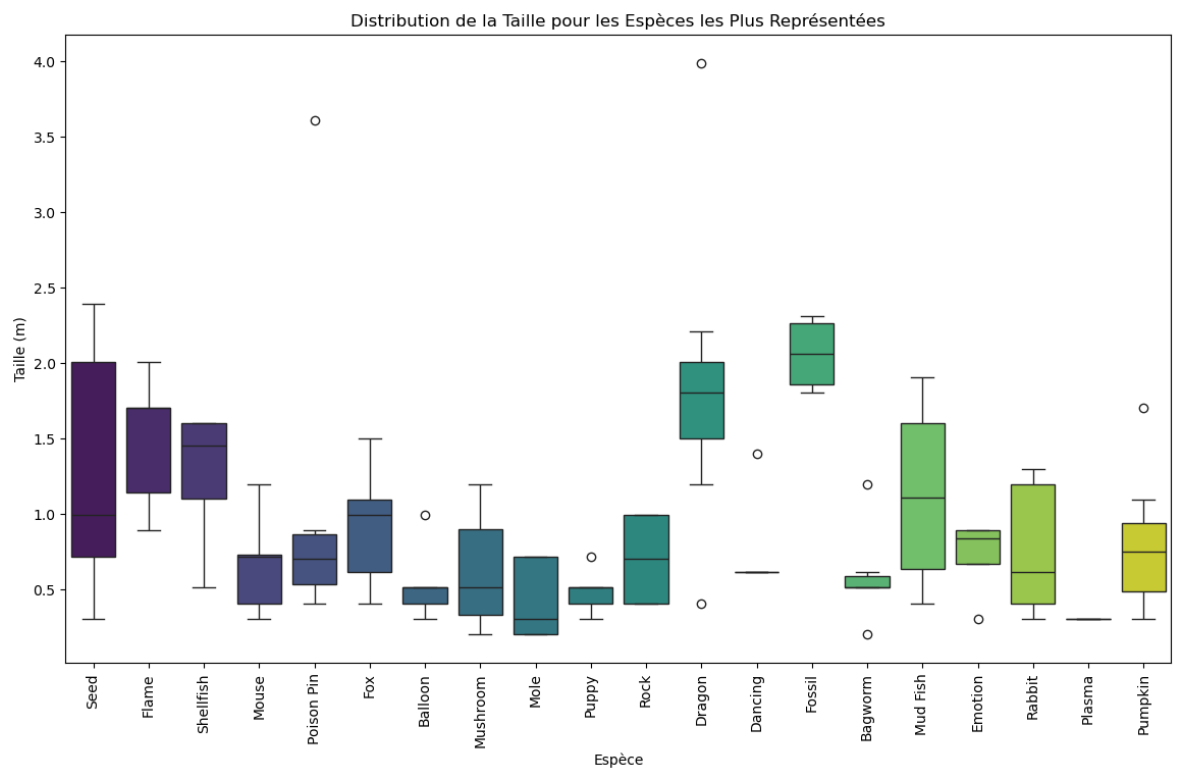
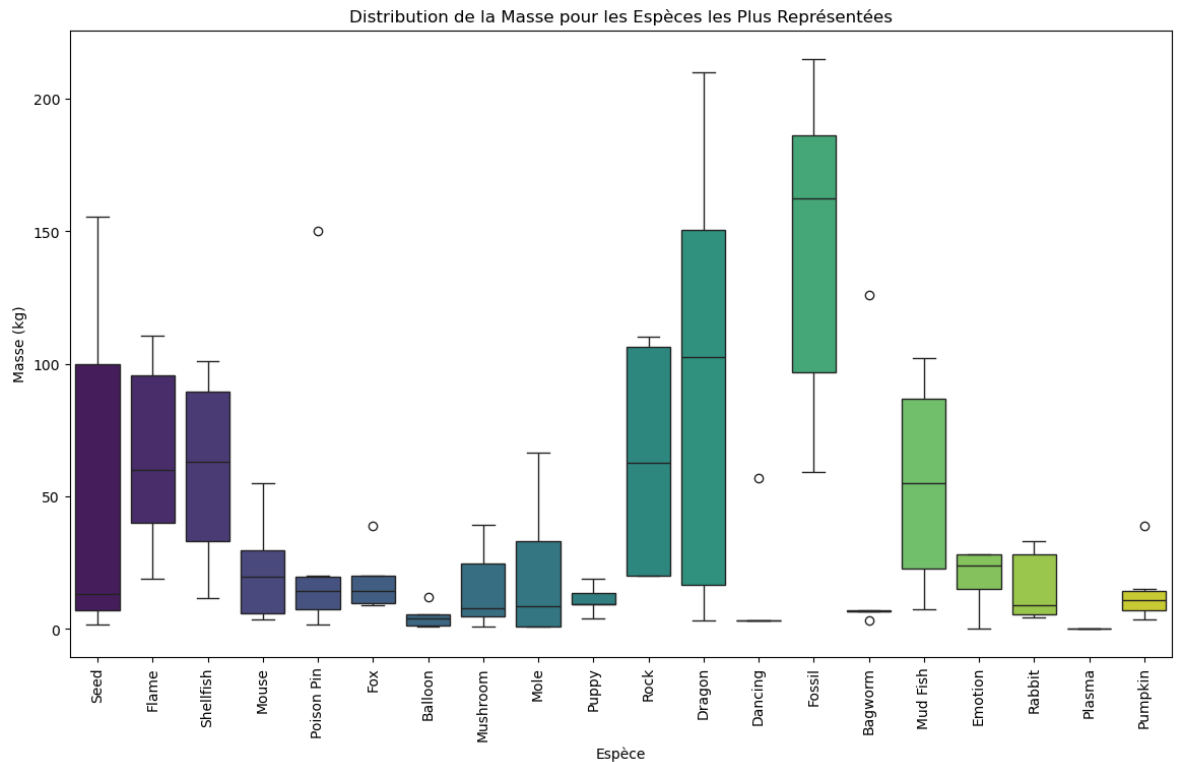
```
In [60]: top_species = dfPokemon['species'].value_counts().nlargest(20).index
         filtered_df = dfPokemon[dfPokemon['species'].isin(top_species)]

# boxplot masse
plt.figure(figsize=(14, 8))
sns.boxplot(x='species', y='weight', data=filtered_df, hue='species', palette='v
```



```
plt.title('Distribution de la Masse pour les Espèces les Plus Représentées')
plt.xlabel('Espèce')
plt.xticks(rotation=90)
plt.ylabel('Masse (kg)')
plt.show()

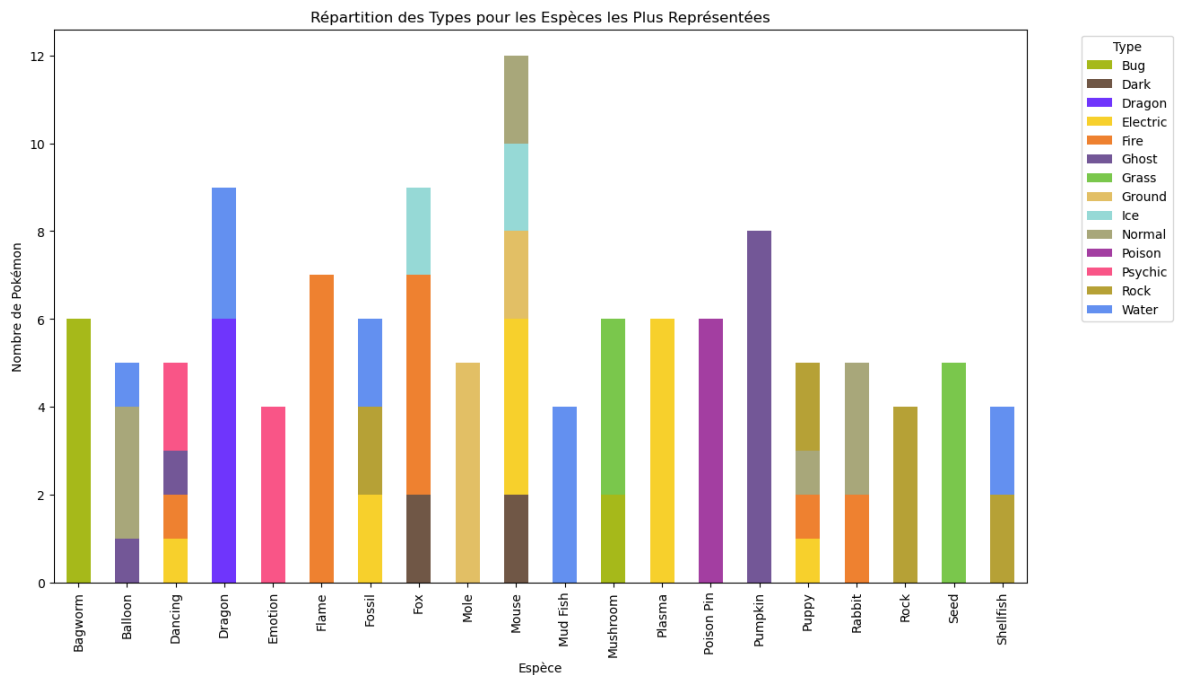
# boxplot taille
plt.figure(figsize=(14, 8))
sns.boxplot(x='species', y='height', data=filtered_df, hue='species', palette='v
plt.title('Distribution de la Taille pour les Espèces les Plus Représentées')
plt.xlabel('Espèce')
plt.xticks(rotation=90)
plt.ylabel('Taille (m)')
plt.show()
```



```
In [61]: type_counts = filtered_df.groupby(['species', 'type_1']).size().unstack(fill_val=0)

colors = [type_color_dict[type_] for type_ in type_counts.columns]

# stack bar plot
type_counts.plot(kind='bar', stacked=True, figsize=(14, 8), color=colors)
plt.title('Répartition des Types pour les Espèces les Plus Représentées')
plt.xlabel('Espèce')
plt.ylabel('Nombre de Pokémon')
plt.xticks(rotation=90)
plt.legend(title='Type', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



```
In [100... combat_stats = filtered_df.groupby('species')[['hp', 'attack', 'defense', 'sp_at

# heatmap
plt.figure(figsize=(14, 8))
sns.heatmap(combat_stats, annot=True, cmap='magma', cbar=False, fmt='g')
plt.title('Statistiques Moyennes de Combat pour les Espèces les Plus Représentée')
plt.xlabel('Statistique de Combat')
plt.ylabel('Espèce')
plt.show()
```

Statistiques Moyennes de Combat pour les Espèces les Plus Représentées						
Esèce	hp	attack	defense	sp_attack	sp_defense	speed
Bagworm	57.5	65.1667	93.3333	55.1667	70.8333	33.1667
Balloon	100	58	39.8	57	38.8	47
Dancing	70	69	69	96.4	74	94.4
Dragon	70.3333	95.7778	82.7778	91.1111	78.3333	81.1111
Emotion	62	65	65	92.5	90	77.5
Flame	74.7143	102.286	78	114.571	88	91.8571
Fossil	86.6667	103.333	88.3333	75	81.6667	90
Fox	56.2222	53.7778	54	73.5556	78.2222	79.5556
Mole	30	79	41	40	55	96.6
Mouse	53.3333	77	66.6667	45.8333	56.6667	79.8333
Mud Fish	80	103.75	80	72.5	80	55
Mushroom	66.3333	79.1667	65	57.5	65	34.1667
Plasma	50	62.5	102	103.333	102	86.8333
Poison Pin	62	64	59.3333	65	55	67.6667
Pumpkin	60.75	79.25	96	51	65	62.5
Puppy	49.8	61	44	39	45	52.2
Rabbit	60	87	64.4	49.4	69.6	97.6
Rock	47.5	87.5	107.5	37.5	37.5	27.5
Seed	59	64.6	69.6	79.4	79	59
Shellfish	62	95.25	103.75	85	83.75	72.75

Interprétation : Beaucoup des variables présentées ici dépendent de l'espèce. Certaines espèces ont bel et bien un profil type, notamment sur leurs caractéristiques (taille,masse), mais d'autres sont beaucoup moins disciplinées et semblent beaucoup plus aléatoires (surtout sur la masse, la taille semble plus typique à chaque espèce).

Il est très intéressant de noter que les espèces s'accordent très bien avec leur type (beaucoup de dragons pour dragon, beaucoup de psy pour émotion, beaucoup d'insectes pour bagworm, etc.).

Les espèces présentent aussi des profils type très intéressants dans leur statistiques de combat pures, il semble que les espèces s'accordent aussi assez bien en moyenne avec leur stats : Roche très défensive, Lapin très rapide, flame très offensive, les chiots un peu faibles dans toutes les catégories, etc.

Réponse : les espèces forment bien des profils type en termes de taille, type, et statistiques. la masse est une variable qui dépend de l'espèce, certaines sont plus disciplinées que d'autres.

Réponse à la question (12) :

On trace un stacked bar plot horizontal pour voir quels sont les types les plus représentatifs de chaque status :

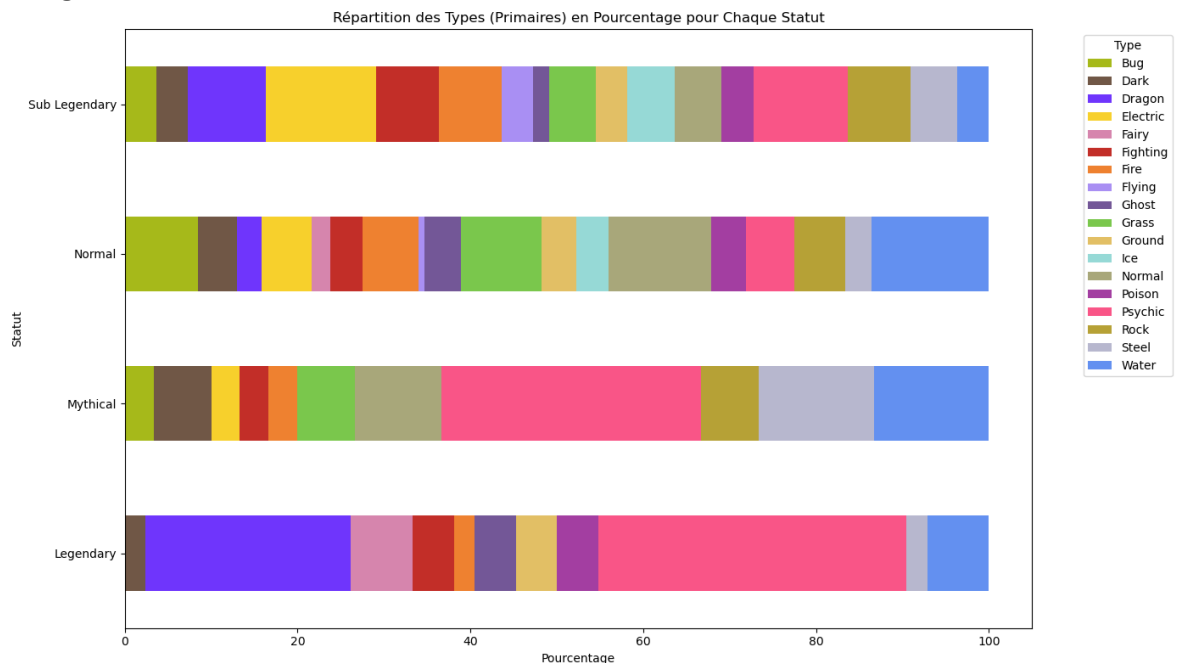
```
In [127... type_counts = dfPokemon.groupby(['status', 'type_1']).size().unstack(fill_value=
colors = [type_color_dict[type_] for type_ in type_counts.columns]

type_percentages = type_counts.div(type_counts.sum(axis=1), axis=0) * 100

plt.figure(figsize=(14, 8))
type_percentages.plot(kind='barh', stacked=True, color=colors, figsize=(14, 8))
plt.title('Répartition des Types (Primaires) en Pourcentage pour Chaque Statut')
```

```
plt.xlabel('Pourcentage')
plt.ylabel('Statut')
plt.legend(title='Type', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()
```

<Figure size 1400x800 with 0 Axes>



On remarque que le type Dragon (encore lui !) est beaucoup plus représenté parmi les Pokémons Légendaires et mythiques

Le type psy est aussi beaucoup plus représentés parmi les Légendaires et mythiques.

Le status normal contient beaucoup plus de pokémons normaux que les autres status.

Ironie du sort, le status mythique contient lui aussi beaucoup de Pokémons de type Normal

Le status semi-légendaire est très équilibré, à la surprise générale, et son type qui le représente le plus est électrique

Réponse : Les status légendaires et mythiques sont beaucoup plus représentés par les types Dragon et Psy, tandis que le status Semi-légendaire n'a pas de type dominant, et forme un tout assez un bien équilibré, comme le status normal.

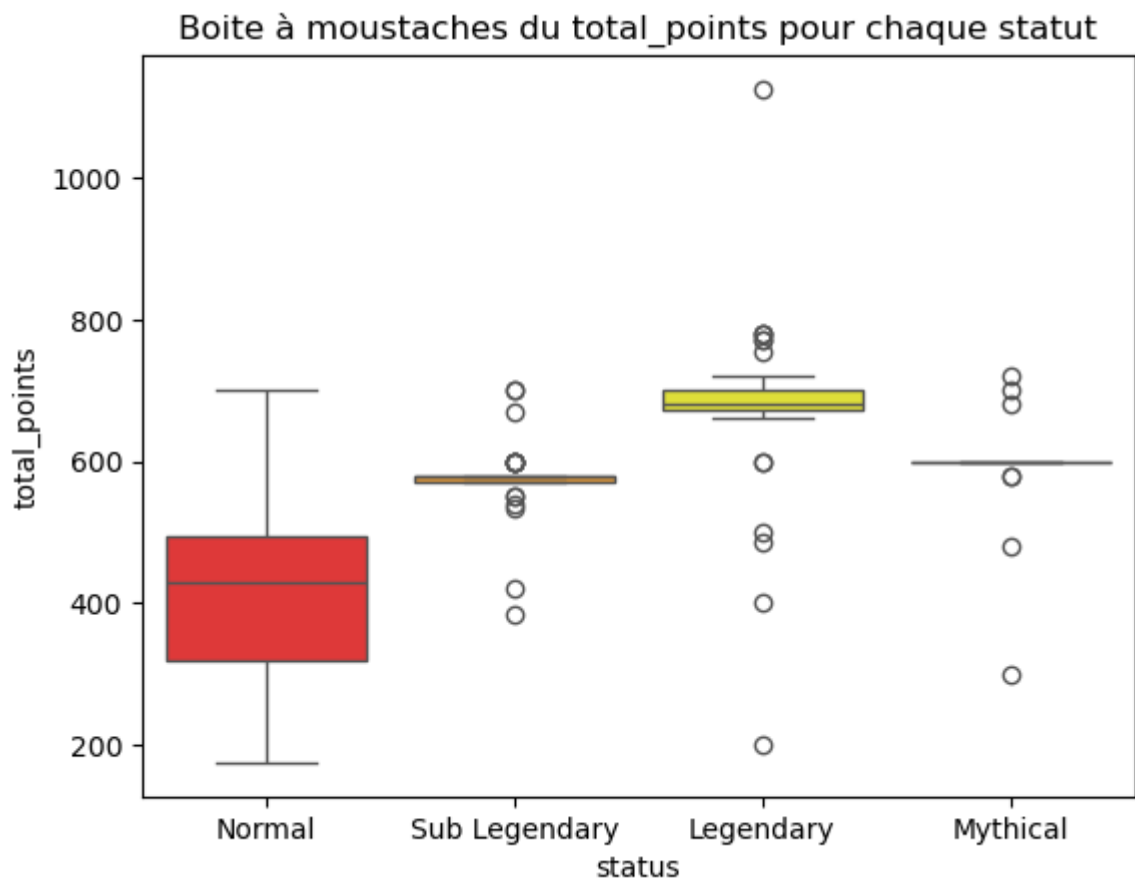
Réponse à la question (11) :

Nous avons émis l'hypothèse que les pokemons avec un nombre de points élevé avaient plus de chances d'etre légendaire.

Comment tester le lien entre la variable total_points et le statut?

On va réaliser une **ANOVA** qui est adaptée ici mais avant on va simplement visualiser la distribution de total_points en fonction du statut :

```
In [236... sns.boxplot(x='status',y='total_points',data=dfPokemon, hue='status', palette=st
plt.title("Boite à moustaches du total_points pour chaque statut")
plt.show()
```



On constate des valeurs plus élevées pour les status semi-légendaire, légendaire et mythique

```
In [167... dfPokemon['status'].value_counts()### beaucoup plus de Pokemons normal dans Le d
```

```
Out[167... status
Normal          918
Sub Legendary    55
Legendary        42
Mythical         30
Name: count, dtype: int64
```

Comme présenté précédemment, on voit que le dataset a beaucoup plus de pokémons normaux que de pokémons non-normaux

```
In [208... def decision(pval):
    if pval<=0.05:### seuil par défaut
        return "On rejette H0 avec un seuil d'erreur de 5% \n"
    else:
        return "On ne rejette pas H0 \n"
```

Posons les hypothèses

- H0: Pas de différences significatives de moyennes entre les status
- H1: il existe au moins un statut qui diffère

```
In [176... groupes = [dfPokemon[dfPokemon['status'] == statut]['total_points'] for statut i
```

```
In [186... f_stat, p_value = f_oneway(*groupes) # On suppose les hypothese pour appliquer le
```

```
In [189... p_value# p-value calculée
```

Out[189... 1.49719602590481e-79

```
In [192... decision(p_value)
```

```
Out[192]: "On rejette H0 avec un seuil d'erreur de 5%"
```

On va compléter l'ANOVA qui ne donne pas les groupes ici les statuts qui diffèrent :

```
In [198... status=dfPokemon['status'].unique()#modalités différentes pour la variable statu
```

```
In [201... paires=list(itertools.combinations(status,2))#toutes les paires de status à test
len(paires)
```

Out[201... 6

```
In [210... for g1,g2 in paires:
    print(g1,g2)# affiche la paire de status tester
    points1=dfPokemon[dfPokemon['status']==g1]['total_points']
    points2=dfPokemon[dfPokemon['status']==g2]['total_points']
    t_stat, p_val = ttest_ind(points1, points2, equal_var=False)###H0 Les moyenn
    print(decision(p_val))
```

Normal Sub Legendary
On ne rejette pas H_0

Normal Legendary
On ne rejette pas H_0

Normal Mythical
On ne rejette pas H_0

Sub Legendary Legendary
On ne rejette pas H_0

Sub Legendary Mythical
On ne rejette pas H_0

Legendary Mythical
On ne rejette pas H_0

l'ANOVA a détectée une différence globale entre les statuts mais on n'a pas identifié les statuts qui diffèrent avec les tests réalisés ci-dessus

```
In [217... # Test de Tukey
tukey = pairwise_tukeyhsd(endog=dfPokemon['total_points'],
                           groups=dfPokemon['status'],
                           alpha=0.05)

print(tukey)
```

Multiple Comparison of Means - Tukey HSD, FWER=0.05

group1	group2	meandiff	p-adj	lower	upper	reject
Legendary	Mythical	-80.6429	0.0056	-143.6062	-17.6795	True
Legendary	Normal	-260.0459	0.0	-301.6079	-218.4839	True
Legendary Sub	Legendary	-98.7823	0.0	-152.7565	-44.808	True
Mythical	Normal	-179.4031	0.0	-228.2715	-130.5346	True
Mythical Sub	Legendary	-18.1394	0.8632	-77.9219	41.6431	False
Normal Sub	Legendary	161.2637	0.0	124.6991	197.8282	True

Les seules groupes qui ne présentent pas de différences sont mythiques et semi-légendaires. La variable total_points permet bien de différencier les status entre eux ce qui est cohérent

Réponse : Oui, plus la variable 'total_points' est élevée, plus le pokémon a de chances de ne pas avoir un status normal. Les Pokémons "non-normaux" ont donc des statistiques plus élevées

Hypothèses Supplémentaires

Dans cette catégorie, on fera quelques tests de khi pour tester l'indépendance entre des variables qualitatives :

Hypothèse : Le type de Pokémon est-il indépendant de son statut ?

```
In [242...] contingency_table = pd.crosstab(dfPokemon['type_1'], dfPokemon['status'])

chi2, p_value, dof, expected = chi2_contingency(contingency_table)

print(f"Chi-squared: {chi2}")
print(f"P-value: {p_value}")
print(f"Degrees of freedom: {dof}")

if p_value < 0.05:
    print("Le type de Pokémon et le statut ne sont pas indépendants.")
else:
    print("Le type de Pokémon et le statut sont indépendants.")
```

Chi-squared: 185.59213518535427

P-value: 3.5082721776551673e-17

Degrees of freedom: 51

Le type de Pokémon et le statut ne sont pas indépendants.

Comme les types et le status ne sont pas indépendants, on peut en conclure que la plupart des types principalement représentés parmi les pokémons non-normaux sont les principaux représentants de leurs types, ce qui montre en toute logique que les types qui sont représentés par les non-normaux sont plus difficiles à capturer.

```
In [246...] contingency_table = pd.crosstab(dfPokemon['type_1'], dfPokemon['generation'])

chi2, p_value, dof, expected = chi2_contingency(contingency_table)
```

```
print(f"Chi-squared: {chi2}")
print(f"P-value: {p_value}")
print(f"Degrees of freedom: {dof}")

if p_value < 0.05:
    print("Le type de Pokémon et la génération ne sont pas indépendants.")
else:
    print("Le type de Pokémon et la génération sont indépendants.")
```

Chi-squared: 234.79471821360502

P-value: 1.3398040350179198e-09

Degrees of freedom: 119

Le type de Pokémon et la génération ne sont pas indépendants.

Comme certains types sont sortis dans des générations ultérieures à la première, ce résultat est logique. Le type Fée n'est apparu qu'à partir de la 4ème génération, et certaines générations ont un type plus représenté que les autres

Conclusion

L'analyse exploratoire menée sur le dataset des Pokémon a permis de mieux comprendre la distribution et les relations entre leurs différentes caractéristiques.

Il apparait clairement que des profils types se forment en fonction de leur type, status, species. Les profils présentés sont bien distincts et chacune des variables apportent un profil unique en leur genre. Les visualisations ont également permis de faire ressortir des tendances non triviales, comme la concentration de certaines espèces ou la domination de certains types dans les catégories supérieures.

Les différents graphiques et tests statistiques on permis de prouver que :

- La masse et la taille des pokémons n'est pas extrêmement corrélée, malgré que significativement liée.
- Les pokémons ayant plus de 600 points de statistiques en tout ne sont pas forcément des légendaires (ou non-normaux).
- Plus le taux de capture est bas, plus il est difficile de capturer un pokémon.
- Les Pokémon plus difficiles à capturer sont en général plus puissants.
- Les Pokémon rapportant plus d'expérience à la victoire d'un combat sont plus puissants.
- Les Pokémon qui rapportent plus d'expérience sont plus difficiles à capturer.
- Les espèces forment des profils type en termes de taille, type, et statistiques. la masse est une variable qui dépend de l'espèce, certaines sont plus disciplinées que d'autres.
- Les types Acier, Psy, Dragon et Vol comportent les Pokémon les plus puissants
- Les types Feu, Acier, Psy, Roche, Dragon sont plus difficiles à capturer.
- Les types dominants dans les status Légendaire et Mythique sont Dragon et Psy, il n'y a pas de type dominant pour les Semi-légendaires.
- Le type de Pokémon et le statut ne sont pas indépendants
- Le type de Pokémon et la génération ne sont pas indépendants

Cette analyse ouvre la voie à des modèles prédictifs plus avancés, en se basant sur les attributs observés pour tenter, par exemple, de prédire le statut d'un Pokémon ou son niveau de puissance général.