pokemon

January 26, 2024

1 Pokemon

In this assessment, you'll read, process, and group CSV data to compute descriptive statistics in two ways for each problem: with the Pandas library and without the Pandas library.

[2]:
```
import pandas as pd
import io

# For prettifying doctest output involving data structures
# See also: https://stackoverflow.com/a/21227671
from pprint import pprint
```

In the *Pokémon* video game series, the player catches **pokemon**, fictional creatures trained to battle each other as part of a sport franchise. For this first task, you'll practice creating your own pokemon-themed CSV dataset in the following format.

[3]:
```
pokemon_box = pd.read_csv("pokemon_box.csv")
pokemon_box
```

[3]:

	id	name	level	personality	type	weakness	atk	def	hp	stage
0	53	Persian	40	mild	normal	fighting	104	116	147	2
1	126	Magmar	44	docile	fire	water	96	83	153	1
2	99	Kingler	33	adamant	water	electric	110	169	29	2
3	57	Primeape	9	lonely	fighting	flying	20	66	43	2
4	3	Venusaur	44	sassy	grass	fire	136	195	92	3
..	…	…	…	…	…	…	…	…	…	…
110	76	Golem	78	hardy	rock	water	65	145	137	
111	116	Horsea	69	mild	water	electric	49	36	45	
112	6	Charizard	89	lax	fire	water	165	100	108	
113	65	Alakazam	33	impish	psychic	dark	67	39	169	
114	120	Staryu	73	hardy	water	electric	30	91	158	

```
..     …
110      3
111      1
112      3
113      3
114      1

[115 rows x 10 columns]
```

- `id` is a unique numeric identifier corresponding to the species of a pokemon.
- `name` is the name of the species of pokemon, such as Bulbasaur.
- `level` is the integer level of the pokemon.
- `personality` is a one-word string describing the personality of the pokemon, such as Jolly.
- `type` is a one-word string describing the type of the pokemon, such as Grass.
- `weakness` is the enemy type that this pokemon is weak toward. Bulbasaur is weak to fire-type pokemon.
- `atk`, `def`, `hp` are integers that indicate the attack power, defense power, and hit points of the pokemon.
- `stage` is an integer that indicates the particular developmental stage of the pokemon.

  Assume the **data is never empty** (there's at least one pokemon), that there's **no missing data** (each pokemon has every attribute), and **pokemon stats can be any non-negative integers, including 0**.

This assessment introduces a new way of validating and testing your data programs by comparing two different approaches to implementing the same function: writing an implementation once using plain Python and again using Pandas. For each programming task below, you'll write and test each function in the same way to build confidence in their correctness and robustness. In addition to the large `pokemon_box` dataset above, we've provided a much smaller `pokemon_test` dataset below.

[4]:
```
pokemon_test = pd.read_csv(io.StringIO("""
id,name,level,personality,type,weakness,atk,def,hp,stage
59,Arcanine,35,impish,fire,water,50,55,90,2
59,Arcanine,35,gentle,fire,water,45,60,80,2
121,Starmie,67,sassy,water,electric,174,56,113,2
131,Lapras,72,lax,water,electric,107,113,29,1
"""))
pokemon_test
```

[4]:

	id	name	level	personality	type	weakness	atk	def	hp	stage
0	59	Arcanine	35	impish	fire	water	50	55	90	2
1	59	Arcanine	35	gentle	fire	water	45	60	80	2
2	121	Starmie	67	sassy	water	electric	174	56	113	2
3	131	Lapras	72	lax	water	electric	107	113	29	1

Note that it's possible to have multiple pokemon that have very similar attributes. In the `pokemon_test` dataset, there are two pokemon named "Arcanine" with the same `id`, `level`, and `type` — differing only in `personality`, `atk`, `def`, and `hp`. Since there's not a clearly unique key to use as an index, we won't define a meaningful index for this assessment.

1.1 Task: Create your own dataset

Before starting your programming tasks, create at least one additional testing dataset below. In total, each function you write should contain 3 tests:

1. One test for the large `pokemon_box` dataset.
2. One test for the small `pokemon_test` dataset.
3. One test for your own `pokemon_mine` dataset below.

[5]:
```
pokemon_mine = pd.read_csv(io.StringIO("""
id,name,level,personality,type,weakness,atk,def,hp,stage
76,Golem,78,hardy,rock,water,65,145,137,3
116,Horsea,69,mild,water,electric,49,36,45,1
53,Persian,40,mild,normal,fighting,104,116,147,2
99,Kingler,33,adamant,water,electric,110,169,29,2
"""))
pokemon_mine
```

[5]:

	id	name	level	personality	type	weakness	atk	def	hp	stage
0	76	Golem	78	hardy	rock	water	65	145	137	3
1	116	Horsea	69	mild	water	electric	49	36	45	1
2	53	Persian	40	mild	normal	fighting	104	116	147	2
3	99	Kingler	33	adamant	water	electric	110	169	29	2

1.2 Task: Species count

Write a function `python_species_count` that takes a list of dictionaries representing the pokemon dataset and returns the number of unique pokemon species in the dataset as determined by the `name` attribute without using Pandas.

Write a function `pandas_species_count` that does the same thing but using a `DataFrame` as input.

Add your test case and a descriptive docstring for both functions.

[11]:
```
def python_species_count(data):
    """
    Calculates the number of unique pokemon species in the dataset

    Args:
    - data (list): a list of dictionaries that contains information for each
                   pokemon in the dataset

    Returns:
    - the number of unique pokemon species in the provided dataset

    >>> python_species_count(pokemon_box.to_records("dict"))
    82
    >>> python_species_count(pokemon_test.to_records("dict"))
    3
    >>> python_species_count(pokemon_mine.to_records("dict"))
```

```
    4
    """
    res = set()
    for list in data:
        res.add(list[2])

    return len(res)
```

[10]:
```
def pandas_species_count(data):
    """
    Calculates the number of unique pokemon species in the dataset

    Args:
    - data (DataFrame): a pandas dataframe that contains pokemon data

    Returns:
    - the number of unique pokemon species in the provided dataset

    >>> pandas_species_count(pokemon_box)
    82
    >>> pandas_species_count(pokemon_test)
    3
    >>> pandas_species_count(pokemon_mine)
    4
    """
    return len(data['name'].unique())
```

1.3 Task: Max level

Write a function `python_max_level` that takes a list of dictionaries representing the pokemon dataset and returns a 2-element tuple for the `(name, level)` of the pokemon with the highest `level` in the dataset. If there are multiple pokemon with the highest `level`, return the pokemon that appears first in the dataset.

Write a function `pandas_max_level` that does the same thing but using a `DataFrame` as input.

Add your test case and a descriptive docstring for both functions.

[9]:
```
def python_max_level(data):
    """
    Finds the pokemon with the highest level as well as the level itself

    Args:
    - data (list): a list of dictionaries that contains information for each
                   pokemon in the dataset

    Returns:
    - a tuple containing the name of the pokemon with the highest level and
```

```
    the level they are at

>>> python_max_level(pokemon_box.to_dict("records"))
('Victreebel', 100)
>>> python_max_level(pokemon_test.to_dict("records"))
('Lapras', 72)
>>> python_max_level(pokemon_mine.to_dict("records"))
('Golem', 78)
"""
max_lvl = -1
name = ''
for dict in data:
    if dict['level'] > max_lvl:
        max_lvl = dict['level']
        name = dict['name']
return tuple((name, max_lvl))
```

[12]:
```
def pandas_max_level(data):
    """
    Finds the pokemon with the highest level as well as the level itself

    Args:
    - data (DataFrame): a pandas dataframe that contains pokemon data

    Returns:
    - a tuple containing the name of the pokemon with the highest level and
      the level they are at

    >>> pandas_max_level(pokemon_box)
    ('Victreebel', 100)
    >>> pandas_max_level(pokemon_test)
    ('Lapras', 72)
    >>> pandas_max_level(pokemon_mine)
    ('Golem', 78)
    """
    max_level = data['level'].max()
    name = data[data['level'] == max_level]['name'].iloc[0]
    return (name, max_level)
```

1.4 Task: Filter range

Write a function `python_filter_range` that takes a list of dictionaries representing the pokemon dataset and two integers: a lower bound (inclusive) and upper bound (exclusive). The function should return a list of the names of pokemon whose `level` fall within the bounds in the same order that they appear in the dataset.

Write a function `pandas_filter_range` that does the same thing but using a `DataFrame` as input. To convert a `Series` to a `list`, use the built-in `list` function as shown below.

```
csv = """
name,age,species
Fido,4,dog
Meowrty,6,cat
Chester,1,dog
Phil,1,axolotl
"""
data = pd.read_csv(io.StringIO(csv))

list(data['name'])
# ['Fido', 'Meowrty', 'Chester', 'Phil']

list(data.loc[1])
# ['Meowrty', 6, 'cat']
```

Add your test case and a descriptive docstring for both functions.

[13]:
```
def python_filter_range(data, lower, upper):
    """
    Finds all pokemon whose level is in between 2 provided bounds inclusive of
    lower and exclusive of upper

    Args:
    - data (list): a list of dictionaries that contains information for each
                   pokemon in the dataset
    - lower (int): the lower level bound
    - uooer (int): the upper level bound

    >>> pprint(python_filter_range(pokemon_box.to_dict("records"), 0, 10))
    ['Primeape',
     'Metapod',
     'Caterpie',
     'Ninetales',
     'Weezing',
     'Tangela',
     'Butterfree',
     'Exeggcute',
     'Arcanine']
    >>> pprint(python_filter_range(pokemon_test.to_dict("records"), 35, 72))
    ['Arcanine', 'Arcanine', 'Starmie']
    >>> pprint(python_filter_range(pokemon_mine.to_dict("records"), 35, 72))
    ['Horsea', 'Persian']
    """
    return [data[i]['name'] for i, j in enumerate(data) if lower <=␣
↪data[i]['level'] < upper]
```

[14]:
```
def pandas_filter_range(data, lower, upper):
    """
```

```
    Finds all pokemon whose level is in between 2 provided bounds inclusive of
    lower and exclusive of upper

    Args:
    - data (DataFrame): a pandas dataframe that contains pokemon data
    - lower (int): the lower level bound
    - uooer (int): the upper level bound

    >>> pprint(pandas_filter_range(pokemon_box, 0, 10))
    ['Primeape',
     'Metapod',
     'Caterpie',
     'Ninetales',
     'Weezing',
     'Tangela',
     'Butterfree',
     'Exeggcute',
     'Arcanine']
    >>> pprint(pandas_filter_range(pokemon_test, 35, 72))
    ['Arcanine', 'Arcanine', 'Starmie']
    >>> pprint(pandas_filter_range(pokemon_mine, 35, 72))
    ['Horsea', 'Persian']
    """
    return list(data[(lower <= data['level']) & (data['level'] <␣
↪upper)]['name'])
```

1.5 Task: Mean attack for type

Write a function `python_mean_attack_for_type` that takes a list of dictionaries representing the pokemon dataset and a `str` representing the pokemon `type`. The function should return the average `atk` for all the pokemon in the dataset with the given `type`. If there are no pokemon of the given `type`, return `None`.

Write a function `pandas_mean_attack_for_type` that does the same thing but using a `DataFrame` as input.

Add your test case and a descriptive docstring for both functions.

[15]:
```
def python_mean_attack_for_type(data, pokemon_type):
    """
    Finds the average attack power for the provided type

    Args:
    - data (list): a list of dictionaries that contains information for each
                   pokemon in the dataset
    - pokemon_type: the pokemon type we are searching the average for

    >>> python_mean_attack_for_type(pokemon_box.to_dict("records"), "water")
```

```
    99.75
    >>> python_mean_attack_for_type(pokemon_test.to_dict("records"), "fire")
    47.5
    >>> python_mean_attack_for_type(pokemon_mine.to_dict("records"), "grass")
    'Type does not exist'
    >>> python_mean_attack_for_type(pokemon_mine.to_dict("records"), "water")
    79.5
    """
    num = 0
    sum = 0
    for dict in data:
        if dict['type'] == pokemon_type:
            sum += dict['atk']
            num += 1
    if num == 0:
        return 'Type does not exist'
    return sum / num
```

[16]:
```
def pandas_mean_attack_for_type(data, pokemon_type):
    """
    Finds the average attack power for the provided type

    Args:
    - data (DataFrame): a pandas dataframe that contains pokemon data
    - pokemon_type: the pokemon type we are searching the average for

    >>> pandas_mean_attack_for_type(pokemon_box, "water")
    99.75
    >>> pandas_mean_attack_for_type(pokemon_test, "fire")
    47.5
    >>> pandas_mean_attack_for_type(pokemon_mine, "fire")
    'Type does not exist'
    >>> pandas_mean_attack_for_type(pokemon_mine, "water")
    79.5
    >>> pandas_mean_attack_for_type(pokemon_mine, "rock")
    65.0
    """
    if not (data['type'] == pokemon_type).any():
        return 'Type does not exist'
    return data[data['type'] == pokemon_type]['atk'].mean()
```

1.6 Task: Count types

Write a function `python_count_types` that takes a list of dictionaries representing the pokemon dataset and returns a dictionary of each pokemon `type` and the number of pokemon of that `type`. The order of entries in the returned dictionary does not matter.

Write a function `pandas_count_types` that does the same thing but using a `DataFrame` as input.

To convert a `Series` to a `dict`, use the built-in `dict` function as shown below.

```
csv = """
name,age,species
Fido,4,dog
Meowrty,6,cat
Chester,1,dog
Phil,1,axolotl
"""
data = pd.read_csv(io.StringIO(csv))

dict(data['name'])
# {0: 'Fido', 1: 'Meowrty', 2: 'Chester', 3: 'Phil'}

dict(data.loc[1])
# {'name': 'Meowrty', 'age': 6, 'species': 'cat'}
```

Add your test case and a descriptive docstring for both functions.

[17]:
```
def python_count_types(data):
    """
    Finds the number of pokemon for each type that exists in the dataset

    Args:
    - data (list): a list of dictionaries that contains information for each
                   pokemon in the dataset

    Returns:
    - a dictionary containing each element as the key and the number of
      pokemon with that type as the value

    >>> pprint(python_count_types(pokemon_box.to_dict("records")))
    {'bug': 3,
     'electric': 1,
     'fairy': 3,
     'fighting': 3,
     'fire': 15,
     'flying': 6,
     'ghost': 3,
     'grass': 17,
     'ground': 5,
     'normal': 10,
     'poison': 12,
     'psychic': 6,
     'rock': 7,
     'water': 24}
    >>> pprint(python_count_types(pokemon_test.to_dict("records")))
    {'fire': 2, 'water': 2}
```

```
    >>> pprint(python_count_types(pokemon_mine.to_dict("records")))
    {'normal': 1, 'rock': 1, 'water': 2}
    """
    res = {}
    for dict in data:
        if dict['type'] in res:
            res[dict['type']] += 1
        else:
            res[dict['type']] = 1
    return res
```

[18]:
```
def pandas_count_types(data):
    """
    Finds the number of pokemon for each type that exists in the dataset

    Args:
    - data (DataFrame): a pandas dataframe that contains pokemon data

    Returns:
    - a dictionary containing each element as the key and the number of
      pokemon with that type as the value
    >>> pprint(pandas_count_types(pokemon_box))
    {'bug': 3,
     'electric': 1,
     'fairy': 3,
     'fighting': 3,
     'fire': 15,
     'flying': 6,
     'ghost': 3,
     'grass': 17,
     'ground': 5,
     'normal': 10,
     'poison': 12,
     'psychic': 6,
     'rock': 7,
     'water': 24}
    >>> pprint(pandas_count_types(pokemon_test))
    {'fire': 2, 'water': 2}
    >>> pprint(pandas_count_types(pokemon_mine))
    {'normal': 1, 'rock': 1, 'water': 2}
    """
    return {type: count for type, count in data['type'].value_counts().items()}
```

1.7 Task: Mean attack per type

Write a function `python_mean_attack_per_type` that takes a list of dictionaries representing the pokemon dataset and returns a dictionary of each pokemon `type` and the average `atk` of pokemon

of that `type`. The order of entries in the returned dictionary does not matter.

Write a function `pandas_mean_attack_per_type` that does the same thing but using a `DataFrame` as input.

Add your test case and a descriptive docstring for both functions.

[19]:
```
def python_mean_attack_per_type(data):
    """
    Finds the average attack power for each type that exists in the dataset

    Args:
    - data (list): a list of dictionaries that contains information for each
                   pokemon in the dataset

    Returns:
    - a dictionary containing each type as a key and the average attack power
      for that type as its value

    >>> pprint(python_mean_attack_per_type(pokemon_box.to_dict("records")))
    {'bug': 25.0,
     'electric': 64.0,
     'fairy': 76.33333333333333,
     'fighting': 99.66666666666667,
     'fire': 99.4,
     'flying': 110.83333333333333,
     'ghost': 88.0,
     'grass': 105.3529411764706,
     'ground': 116.6,
     'normal': 108.0,
     'poison': 121.75,
     'psychic': 114.83333333333333,
     'rock': 84.85714285714286,
     'water': 99.75}
    >>> pprint(python_mean_attack_per_type(pokemon_test.to_dict("records")))
    {'fire': 47.5, 'water': 140.5}
    >>> pprint(python_mean_attack_per_type(pokemon_mine.to_dict("records")))
    {'normal': 104.0, 'rock': 65.0, 'water': 79.5}
    """
    sum = {}
    count = {}
    for dict in data:
        if dict['type'] in sum:
            sum[dict['type']] += dict['atk']
            count[dict['type']] += 1
        else:
            sum[dict['type']] = dict['atk']
            count[dict['type']] = 1
```

```
    return {type: sum[type]/count[type] for type, avg in zip(sum, count)}
```

[20]:
```
def pandas_mean_attack_per_type(data):
    """
    Finds the average attack power for each type that exists in the dataset

    Args:
    - data (DataFrame): a pandas dataframe that contains pokemon data

    Returns:
    - a dictionary containing each type as a key and the average attack power
      for that type as its value


    >>> pprint(pandas_mean_attack_per_type(pokemon_box))
    {'bug': 25.0,
     'electric': 64.0,
     'fairy': 76.33333333333333,
     'fighting': 99.66666666666667,
     'fire': 99.4,
     'flying': 110.83333333333333,
     'ghost': 88.0,
     'grass': 105.3529411764706,
     'ground': 116.6,
     'normal': 108.0,
     'poison': 121.75,
     'psychic': 114.83333333333333,
     'rock': 84.85714285714286,
     'water': 99.75}
    >>> pprint(pandas_mean_attack_per_type(pokemon_test))
    {'fire': 47.5, 'water': 140.5}
    >>> pprint(pandas_mean_attack_per_type(pokemon_mine))
    {'normal': 104.0, 'rock': 65.0, 'water': 79.5}
    """
    return {type: data[data['type'] == type]['atk'].mean() for type in␣
↪data['type']}
```

1.8 Testing

[21]:
```
import doctest
doctest.testmod()
```

[21]:
```
TestResults(failed=0, attempted=39)
```