Efficiently combining, counting, and iterating

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Scientific Software Technical Trainer, Enthought

datacamp

Pokémon Overview

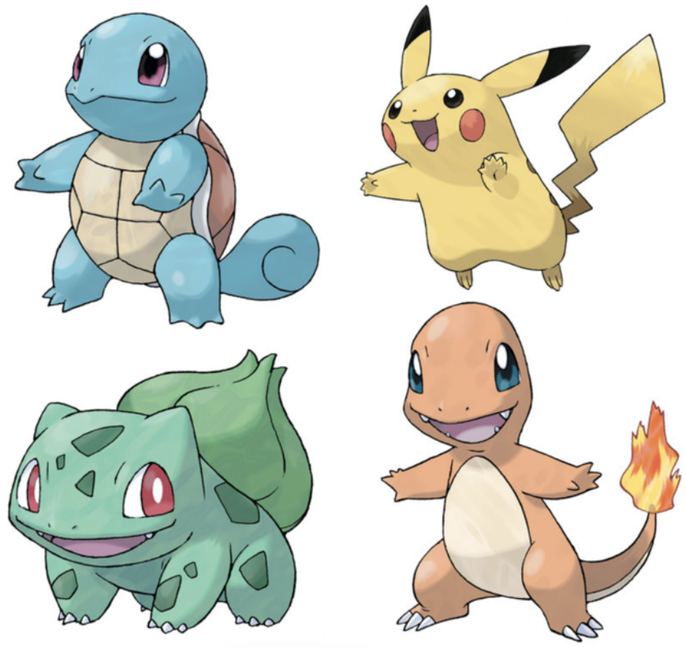
- Trainers (collect Pokémon)

Trainer

Pokémon Overview

- Pokémon (fictional animal characters)

Trainer

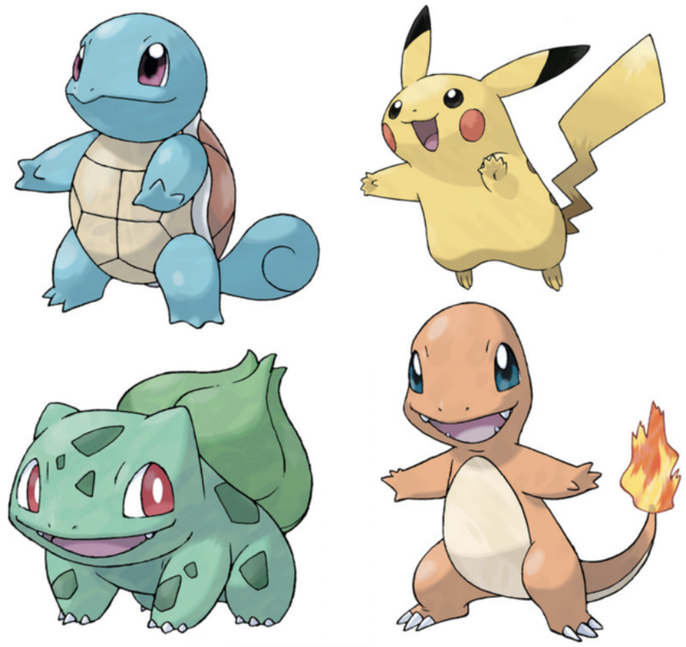


Pokémon

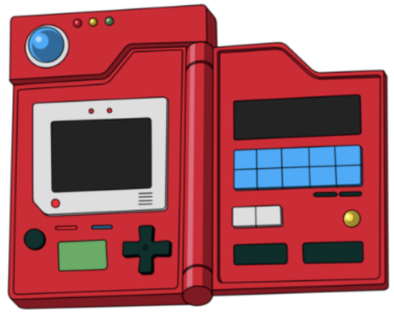
Pokémon Overview

- Pokédex (stores captured Pokémon)

Trainer



Pokémon



Pokédex

Pokémon Description

Squirtle is a Water type Pokémon introduced in Generation 1. It is known as the 'Tiny Turtle Pokémon'.

Pokédex data

National №	**007**
Type	WATER
Legendary	False

Base stats

HP	44	
Attack	48	
Defense	65	
Sp. Atk	50	
Sp. Def	64	
Speed	43	
Total	**314**	

Pokémon Description

Squirtle is a Water type Pokémon introduced in Generation 1. It is known as the 'Tiny Turtle Pokémon'.

Pokédex data

National №	**007**
Type	WATER
Legendary	False

Base stats

HP	44	
Attack	48	
Defense	65	
Sp. Atk	50	
Sp. Def	64	
Speed	43	
Total	**314**	

Pokémon Description

Squirtle is a Water type Pokémon introduced in Generation 1. It is known as the 'Tiny Turtle Pokémon'.

Pokédex data

National №	007
Type	WATER
Legendary	False

Base stats

HP	44	
Attack	48	
Defense	65	
Sp. Atk	50	
Sp. Def	64	
Speed	43	
Total	**314**	

Pokémon Description

Squirtle is a Water type Pokémon introduced in Generation 1. It is known as the 'Tiny Turtle Pokémon'.

Pokédex data

National №	007
Type	WATER
Legendary	False

Base stats

HP	44	
Attack	48	
Defense	65	
Sp. Atk	50	
Sp. Def	64	
Speed	43	
Total	**314**	

Combining objects

```
names = ['Bulbasaur', 'Charmander', 'Squirtle']
hps = [45, 39, 44]
```

```
combined = []

for i,pokemon in enumerate(names):
    combined.append((pokemon, hps[i]))

print(combined)
```

```
[('Bulbasaur', 45), ('Charmander', 39), ('Squirtle', 44)]
```

Combining objects with zip

```
names = ['Bulbasaur', 'Charmander', 'Squirtle']
hps = [45, 39, 44]
```

```
combined_zip = zip(names, hps)
print(type(combined_zip))
```

```
<class 'zip'>
```

```
combined_zip_list = [*combined_zip]

print(combined_zip_list)
```

```
[('Bulbasaur', 45), ('Charmander', 39), ('Squirtle', 44)]
```

10. Combining objects with zip
But Python's built-in function zip provides a more elegant solution. The name "zip" describes how this function combines objects like a zipper on a jacket (making two separate things become one). zip returns a zip object that must be unpacked into a list and printed to see the contents. Each item is a tuple of elements from the original lists.

The collections module

- Part of Python's Standard Library (built-in module)
- Specialized container datatypes
  - Alternatives to general purpose dict, list, set, and tuple
- Notable:
  - `namedtuple` : tuple subclasses with named fields
  - `deque` : list-like container with fast appends and pops
  - `Counter` : dict for counting hashable objects
  - `OrderedDict` : dict that retains order of entries
  - `defaultdict` : dict that calls a factory function to supply missing values

The collections module

- Part of Python's Standard Library (built-in module)
- Specialized container datatypes
  - Alternatives to general purpose dict, list, set, and tuple
- Notable:
  - `namedtuple` : tuple subclasses with named fields
  - `deque` : list-like container with fast appends and pops
  - **`Counter` : dict for counting hashable objects**
  - `OrderedDict` : dict that retains order of entries
  - `defaultdict` : dict that calls a factory function to supply missing values

11. The collections module
Python comes with a number of efficient built-in modules. The collections module contains specialized datatypes that can be used as alternatives to standard dictionaries, lists, sets, and tuples.

12. The collections module
A few notable specialized datatypes are listed here. Let's dig deeper into the Counter object.

Counting with loop

```
# Each Pokémon's type (720 total)
poke_types = ['Grass', 'Dark', 'Fire', 'Fire', ...]
type_counts = {}
for poke_type in poke_types:
    if poke_type not in type_counts:
        type_counts[poke_type] = 1
    else:
        type_counts[poke_type] += 1
print(type_counts)
```

13. Counting with loop

Our Pokémon dataset describes 720 characters. Here is a list of each Pokémon's corresponding type. We'd like to create a dictionary where each key is a Pokémon type, and each value is the count of characters that belong to that type. If the poke_type is not in the dictionary, we create a new key and initialize its count value as one. If the poke_type is already in the dictionary, we update the count by one.

```
{'Rock': 41, 'Dragon': 25, 'Ghost': 20, 'Ice': 23, 'Poison': 28, 'Grass': 64,
 'Flying': 2, 'Electric': 40, 'Fairy': 17, 'Steel': 21, 'Psychic': 46, 'Bug': 65,
 'Dark': 28, 'Fighting': 25, 'Ground': 30, 'Fire': 48,'Normal': 92, 'Water': 105}
```

collections.Counter()

```
# Each Pokémon's type (720 total)
poke_types = ['Grass', 'Dark', 'Fire', 'Fire', ...]
from collections import Counter
type_counts = Counter(poke_types)
print(type_counts)
```

14. collections.Counter()
Using Counter from the collections module is a more efficient approach. Just import Counter and provide the object to be counted. No need for a loop! Counter returns a Counter dictionary of key-value pairs. When printed, it's ordered by highest to lowest counts. If comparing runtime times, we'd see that using Counter takes half the time as the standard dictionary approach!

```
Counter({'Water': 105, 'Normal': 92, 'Bug': 65, 'Grass': 64, 'Fire': 48,
         'Psychic': 46, 'Rock': 41, 'Electric': 40, 'Ground': 30,
         'Poison': 28, 'Dark': 28, 'Dragon': 25, 'Fighting': 25, 'Ice': 23,
         'Steel': 21, 'Ghost': 20, 'Fairy': 17, 'Flying': 2})
```

The itertools module

- Part of Python's Standard Library (built-in module)
- Functional tools for creating and using iterators
- Notable:
  - Infinite iterators: `count` , `cycle` , `repeat`
  - Finite iterators: `accumulate` , `chain` , `zip_longest` , etc.
  - Combination generators: `product` , `permutations` , `combinations`

15. The itertools module
Another built-in module, itertools, contains functional tools for working with iterators.
A subset of these tools is listed here.

The itertools module

- Part of Python's Standard Library (built-in module)
- Functional tools for creating and using iterators
- Notable:
  - Infinite iterators: `count` , `cycle` , `repeat`
  - Finite iterators: `accumulate` , `chain` , `zip_longest` , etc.
  - **Combination generators:** **`product`** , **`permutations`** , **`combinations`**

16. The itertools module
We'll focus on one piece of this module: the combinatoric generators. These generators efficiently yield Cartesian products, permutations, and combinations of objects. Let's explore an example.

Combinations with loop

```
poke_types = ['Bug', 'Fire', 'Ghost', 'Grass', 'Water']
combos = []

for x in poke_types:
    for y in poke_types:
        if x == y:
            continue
        if ((x,y) not in combos) & ((y,x) not in combos):
            combos.append((x,y))
print(combos)
```

17. Combinations with loop
Suppose we want to gather all combination pairs of Pokémon types possible. We can do this with a **nested for loop** that iterates over the poke_types list twice. Notice that a conditional statement is used to skip pairs having the same type twice. For example, if x is 'Bug' and y is 'Bug', we want to skip this pair. Since we're interested in combinations (where order doesn't matter), another statement is used to ensure either order of the pair doesn't already exist within the combos list before appending it. For example, the pair ('Bug', 'Fire') is the same as the pair ('Fire', 'Bug'). We want one of these pairs, not both.

```
[('Bug', 'Fire'), ('Bug', 'Ghost'), ('Bug', 'Grass'), ('Bug', 'Water'),
 ('Fire', 'Ghost'), ('Fire', 'Grass'), ('Fire', 'Water'),
 ('Ghost', 'Grass'), ('Ghost', 'Water'), ('Grass', 'Water')]
```

itertools.combinations()

```
poke_types = ['Bug', 'Fire', 'Ghost', 'Grass', 'Water']
from itertools import combinations
combos_obj = combinations(poke_types, 2)
print(type(combos_obj))
```

```
<class 'itertools.combinations'>
```

```
combos = [*combos_obj]
print(combos)
```

```
[('Bug', 'Fire'), ('Bug', 'Ghost'), ('Bug', 'Grass'), ('Bug', 'Water'),
 ('Fire', 'Ghost'), ('Fire', 'Grass'), ('Fire', 'Water'),
 ('Ghost', 'Grass'), ('Ghost', 'Water'), ('Grass', 'Water')]
```

18. itertools.combinations()

The combinations generator from itertools provides a more efficient solution. First, we import combinations and then create a combinations object by providing the poke_types list and the length of combinations we desire. combinations returns a combinations object, which we unpack into a list and print to see the result. **If comparing runtimes, we'd see using combinations is significantly faster than the nested loop.**

Let's practice!

WRITING EFFICIENT PYTHON CODE

Set theory

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Scientific Software Technical Trainer, Enthought

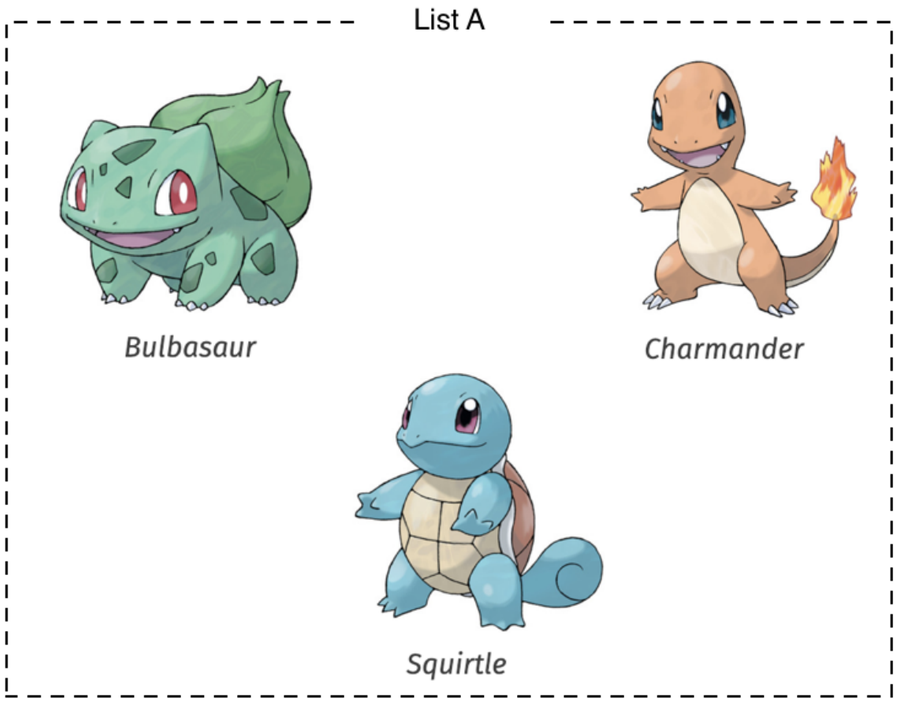
Set theory

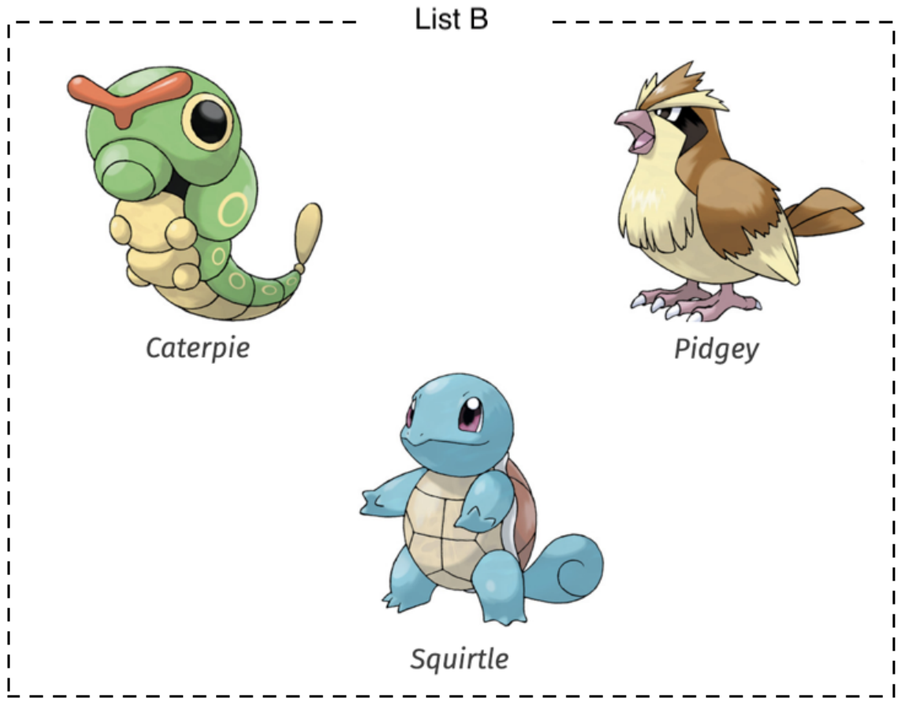
- Branch of Mathematics applied to collections of objects
  - i.e., `sets`
- Python has built-in `set` datatype with accompanying methods:
  - `intersection()` : all elements that are in both sets
  - `difference()` : all elements in one set but not the other
  - `symmetric_difference()` : all elements in exactly one set
  - `union()` : all elements that are in either set
- Fast membership testing
  - Check if a value exists in a sequence or not
  - Using the `in` operator

In this lesson, we'll show that using the in operator with a set is much faster than using it with a list or tuple.

Comparing objects with loops

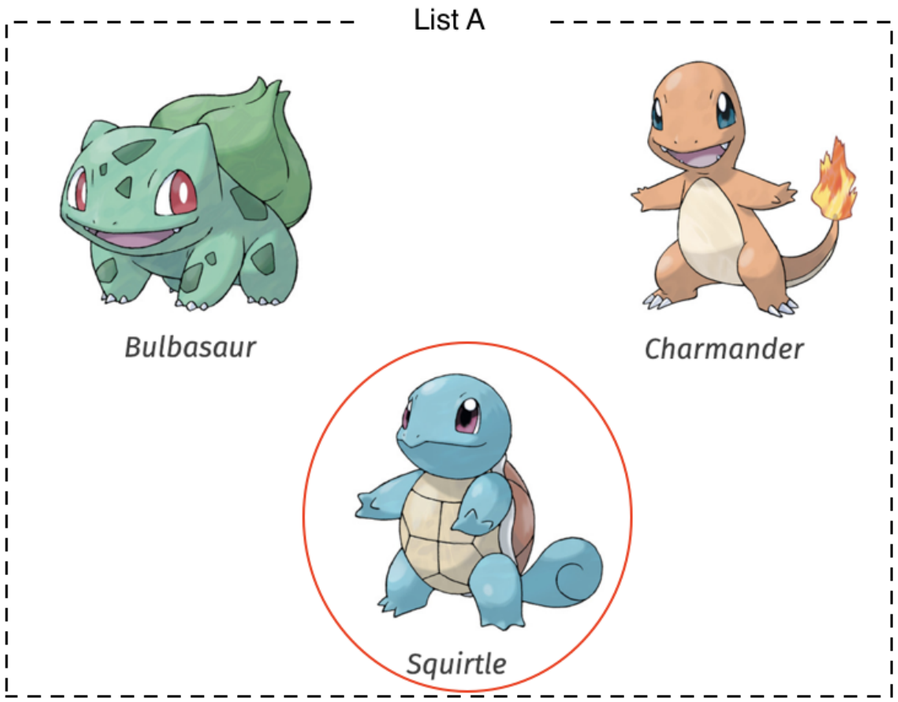
```
list_a = ['Bulbasaur', 'Charmander', 'Squirtle']
list_b = ['Caterpie', 'Pidgey', 'Squirtle']
```

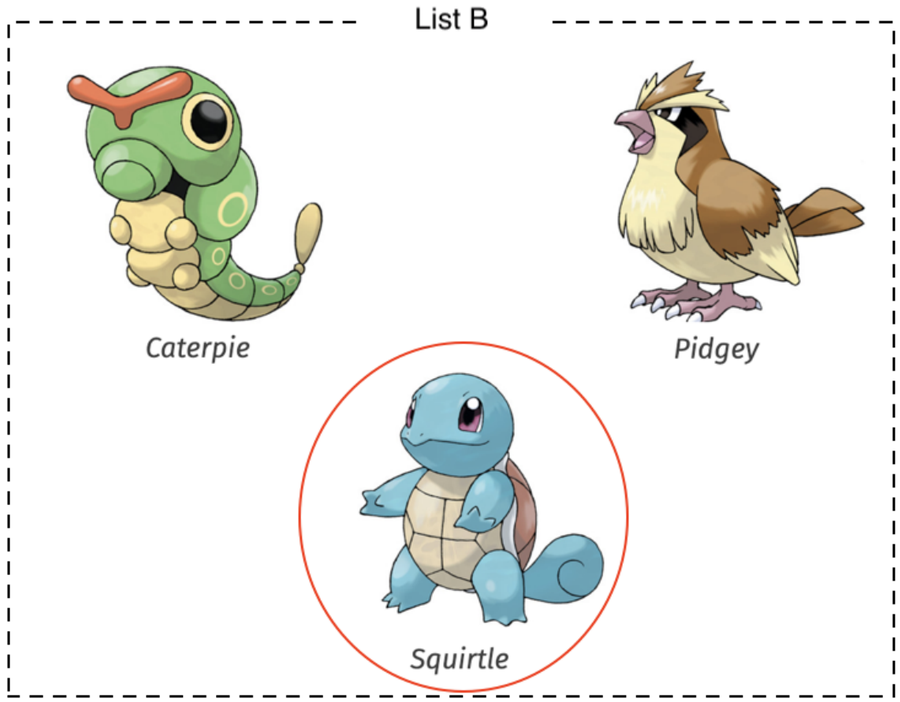




Comparing objects with loops

```
list_a = ['Bulbasaur', 'Charmander', 'Squirtle']
list_b = ['Caterpie', 'Pidgey', 'Squirtle']
```





```
list_a = ['Bulbasaur', 'Charmander', 'Squirtle']
list_b = ['Caterpie', 'Pidgey', 'Squirtle']
```

```
in_common = []

for pokemon_a in list_a:
    for pokemon_b in list_b:
        if pokemon_a == pokemon_b:
            in_common.append(pokemon_a)

print(in_common)
```

5. Comparing objects with loops
We could use a nested for loop to compare each item in list_a to each item in list_b and collect only those items that appear in both lists. **But, iterating over each item in both lists is extremely inefficient.**

```
['Squirtle']
```

```
list_a = ['Bulbasaur', 'Charmander', 'Squirtle']
list_b = ['Caterpie', 'Pidgey', 'Squirtle']
```

6. Comparing objects with set theory
Instead, we should use Python's set data type to compare these lists. By converting each list into a set, we can use the dot-intersection method to collect the Pokémon shared between the two sets. One simple line of code and no need for a loop!

```
set_a = set(list_a)
print(set_a)
```

```
{'Bulbasaur', 'Charmander', 'Squirtle'}
```

```
set_b = set(list_b)
print(set_b)
```

```
{'Caterpie', 'Pidgey', 'Squirtle'}
```

```
set_a.intersection(set_b)
```

```
{'Squirtle'}
```

Efficiency gained with set theory

```
%%timeit
in_common = []

for pokemon_a in list_a:
    for pokemon_b in list_b:
        if pokemon_a == pokemon_b:
            in_common.append(pokemon_a)
```

```
601 ns ± 17.1 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
%timeit in_common = set_a.intersection(set_b)
```

```
137 ns ± 3.01 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

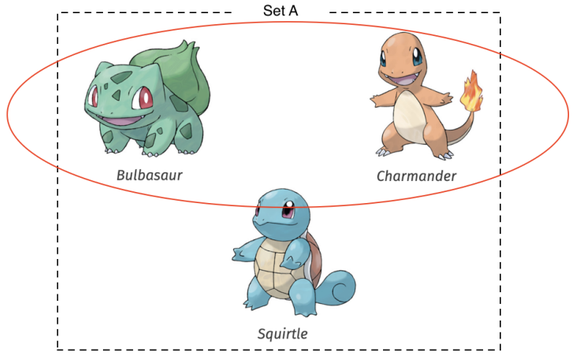
Set method: difference

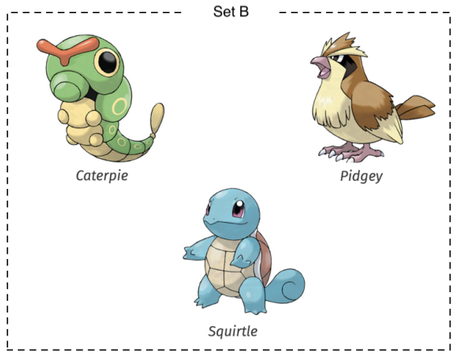
```
set_a = {'Bulbasaur', 'Charmander', 'Squirtle'}
set_b = {'Caterpie', 'Pidgey', 'Squirtle'}
```

```
set_a.difference(set_b)
```

To gather Pokémon that exist in set_a but not in set_b, use set_a-dot-difference(set_b).

```
{'Bulbasaur', 'Charmander'}
```





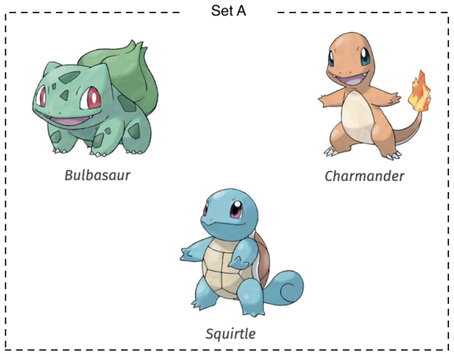
Set method: difference

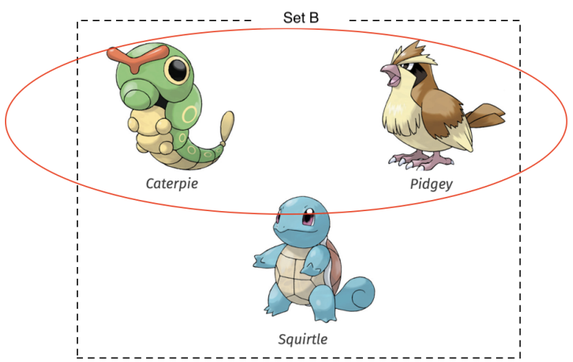
```
set_a = {'Bulbasaur', 'Charmander', 'Squirtle'}
set_b = {'Caterpie', 'Pidgey', 'Squirtle'}
```

```
set_b.difference(set_a)
```

9. Set method: difference
If we want the Pokémon in set_b, but not in set_a, we use set_b-dot-difference(set_a).

```
{'Caterpie', 'Pidgey'}
```





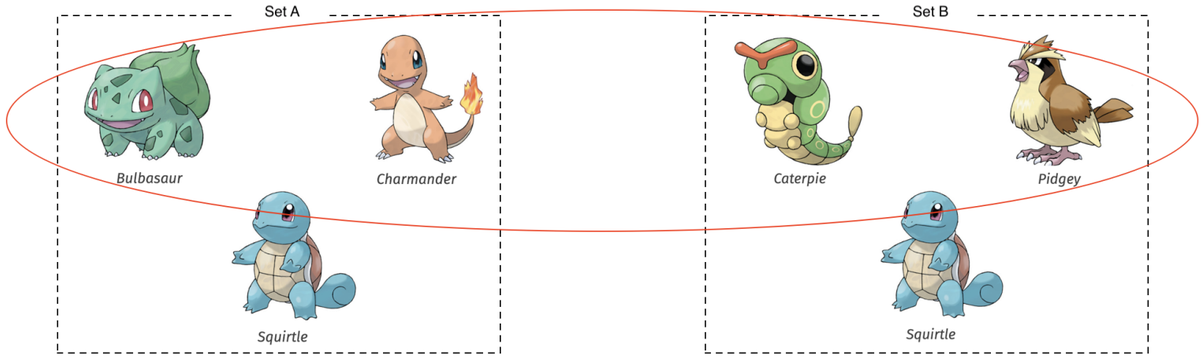
Set method: symmetric difference

```
set_a = {'Bulbasaur', 'Charmander', 'Squirtle'}
set_b = {'Caterpie', 'Pidgey', 'Squirtle'}
```

```
set_a.symmetric_difference(set_b)
```

10. Set method: symmetric difference
To collect Pokémon that exist in exactly one of the sets (but not both), we can use a method called the symmetric difference.

```
{'Bulbasaur', 'Caterpie', 'Charmander', 'Pidgey'}
```



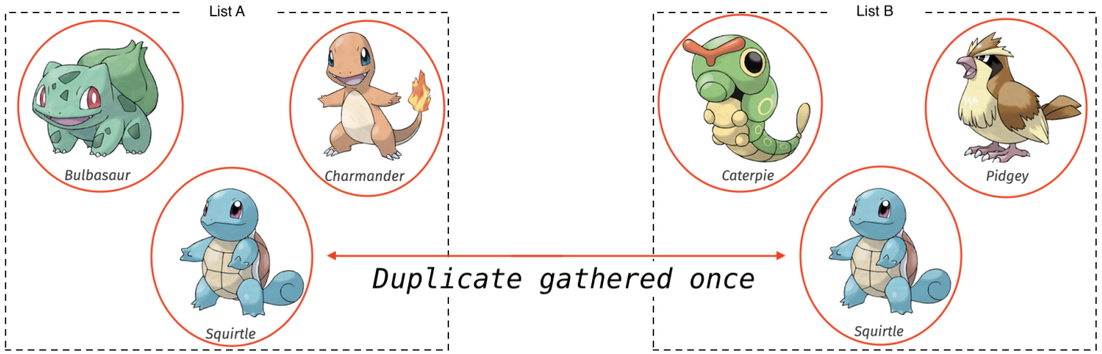
Set method: union

```
set_a = {'Bulbasaur', 'Charmander', 'Squirtle'}
set_b = {'Caterpie', 'Pidgey', 'Squirtle'}
```

```
set_a.union(set_b)
```

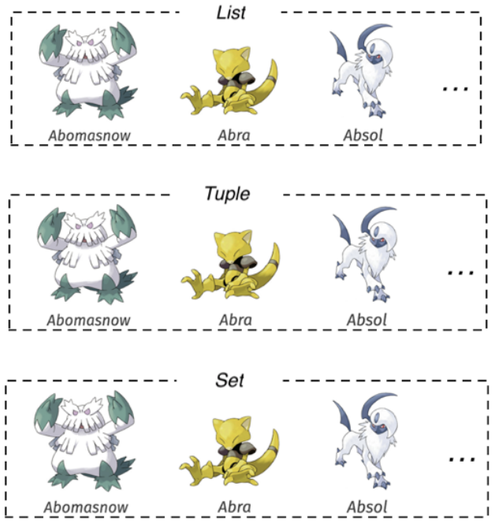
11. Set method: union
Finally, we can combine these sets using the dot-union method. This collects all of the unique Pokémon that appear in either or both sets.

```
{'Bulbasaur', 'Caterpie', 'Charmander', 'Pidgey', 'Squirtle'}
```



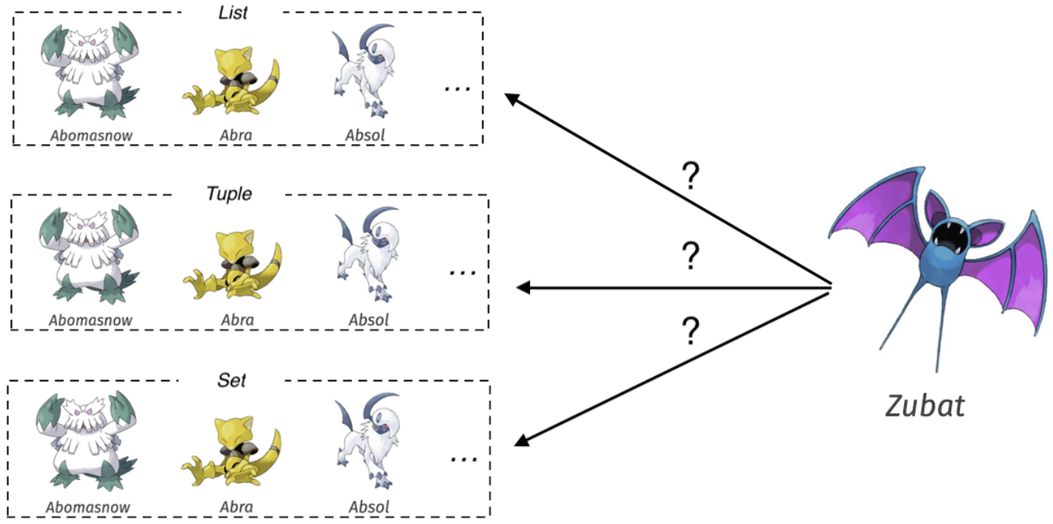
Membership testing with sets

```
# The same 720 total Pokémon in each data structure
names_list  = ['Abomasnow', 'Abra', 'Absol', ...]
names_tuple = ('Abomasnow', 'Abra', 'Absol', ...)
names_set   = {'Abomasnow', 'Abra', 'Absol', ...}
```



Membership testing with sets

```
# The same 720 total Pokémon in each data structure
names_list  = ['Abomasnow', 'Abra', 'Absol', ...]
names_tuple = ('Abomasnow', 'Abra', 'Absol', ...)
names_set   = {'Abomasnow', 'Abra', 'Absol', ...}
```



```
names_list  = ['Abomasnow', 'Abra', 'Absol', ...]
names_tuple = ('Abomasnow', 'Abra', 'Absol', ...)
names_set   = {'Abomasnow', 'Abra', 'Absol', ...}
```

```
%timeit 'Zubat' in names_list
```

```
7.63 µs ± 211 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
%timeit 'Zubat' in names_tuple
```

```
7.6 µs ± 394 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
%timeit 'Zubat' in names_set
```

```
37.5 ns ± 1.37 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Uniques with sets

```
# 720 Pokémon primary types corresponding to each Pokémon
primary_types = ['Grass', 'Psychic', 'Dark', 'Bug', ...]
```

```
unique_types = []

for prim_type in primary_types:
    if prim_type not in unique_types:
        unique_types.append(prim_type)

print(unique_types)
```

15. Uniques with sets

One final efficiency gain when using sets comes from the definition of set itself. A set is defined as a collection of distinct elements. Thus, we can use a set to collect unique items from an existing object. Let's revisit the primary_types list, which contains the primary types of each Pokémon. If we wanted to collect the unique Pokémon types within this list, we could write a for loop to iterate over the list, and only append the Pokémon types that haven't already been added to the unique_types list.

```
['Grass', 'Psychic', 'Dark', 'Bug', 'Steel', 'Rock', 'Normal',
 'Water', 'Dragon', 'Electric', 'Poison', 'Fire', 'Fairy', 'Ice',
 'Ground', 'Ghost', 'Fighting', 'Flying']
```

Uniques with sets

```
# 720 Pokémon primary types corresponding to each Pokémon
primary_types = ['Grass', 'Psychic', 'Dark', 'Bug', ...]
```

```
unique_types_set = set(primary_types)
```

16. Uniques with sets
Using a set makes this much easier. All we have to do is convert the primary_types list into a set, and we have our solution: a set of distinct Pokémon types.

```
print(unique_types_set)
```

```
{'Grass', 'Psychic', 'Dark', 'Bug', 'Steel', 'Rock', 'Normal',
 'Water', 'Dragon', 'Electric', 'Poison', 'Fire', 'Fairy', 'Ice',
 'Ground', 'Ghost', 'Fighting', 'Flying'}
```

Let's practice set theory!

WRITING EFFICIENT PYTHON CODE

Eliminating loops

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Scientific Software Technical Trainer, Enthought

Looping in Python

- Looping patterns:
  - `for` loop: iterate over sequence piece-by-piece
  - `while` loop: repeat loop as long as condition is met
  - "nested" loops: use one loop inside another loop
  - Costly!

Benefits of eliminating loops

- Fewer lines of code
- Better code readability
  - "Flat is better than nested"
- Efficiency gains

Eliminating loops with built-ins

```
# List of HP, Attack, Defense, Speed
poke_stats = [
    [90,  92, 75, 60],
    [25,  20, 15, 90],
    [65, 130, 60, 75],
    ...
]
```

Abomasnow

Base stats

HP	90
Attack	92
Defense	75
Sp. Atk	92
Sp. Def	85
Speed	60
Total	**494**

Abra

Base stats

HP	25
Attack	20
Defense	15
Sp. Atk	105
Sp. Def	55
Speed	90
Total	**310**

Absol

Base stats

HP	65
Attack	130
Defense	60
Sp. Atk	75
Sp. Def	60
Speed	75
Total	**465**

```
# List of HP, Attack, Defense, Speed
poke_stats = [
    [90,  92, 75, 60],
    [25,  20, 15, 90],
    [65, 130, 60, 75],
    ...
]
# For loop approach
totals = []
for row in poke_stats:
    totals.append(sum(row))
# List comprehension
totals_comp = [sum(row) for row in poke_stats]
# Built-in map() function
totals_map = [*map(sum, poke_stats)]
```

5. Eliminating loops with built-ins
We want to do a simple sum of each of these rows in order to collect the total stats for each Pokémon. If we were to use a loop to calculate row sums, we would have to iterate over each row and append the row's sum to the totals list. We can accomplish the same task, in fewer lines of code, with a list comprehension. Or, we could use the built-in map function that we've discussed previously.

6. Eliminating loops with built-ins
Each of these approaches will return the same list, but using a list comprehension or the map function takes one line of code, and has a faster runtime.

```
%%timeit
totals = []
for row in poke_stats:
    totals.append(sum(row))
```

```
140 µs ± 1.94 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
%timeit totals_comp = [sum(row) for row in poke_stats]
```

```
114 µs ± 3.55 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
%timeit totals_map = [*map(sum, poke_stats)]
```

```
95 µs ± 2.94 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Eliminating loops with built-in modules

```
poke_types = ['Bug', 'Fire', 'Ghost', 'Grass', 'Water']
```

```
# Nested for loop approach
combos = []
for x in poke_types:
    for y in poke_types:
        if x == y:
            continue
        if ((x,y) not in combos) & ((y,x) not in combos):
            combos.append((x,y))
```

```
# Built-in module approach
from itertools import combinations
combos2 = [*combinations(poke_types, 2)]
```

Eliminate loops with NumPy

```
# Array of HP, Attack, Defense, Speed
import numpy as np

poke_stats = np.array([
    [90,  92, 75, 60],
    [25,  20, 15, 90],
    [65, 130, 60, 75],
    ...
])
```

Eliminate loops with NumPy

```
avgs = []
for row in poke_stats:
    avg = np.mean(row)
    avgs.append(avg)
print(avgs)
```

```
[79.25, 37.5, 82.5, ...]
```

8. Eliminate loops with NumPy
Another powerful technique for eliminating loops is to use the NumPy package. Suppose we had the same collection of statistics we used in a previous example but stored in a NumPy array instead of a list of lists.

```
avgs_np = poke_stats.mean(axis=1)
print(avgs_np)
```

```
[ 79.25  37.5   82.5  ...]
```

9. Eliminate loops with NumPy
We'd like to collect the average stat value for each Pokémon (or row) in our array. We could use a loop to iterate over the array and collected the row averages. But, NumPy arrays allow us to perform calculations on entire arrays all at once. Here, we use the dot-mean method and specifying an axis equal to 1 to calculate the mean for each row (meaning we calculate an average across the column values). This eliminates the need for a loop and is much more efficient.

Eliminate loops with NumPy

```
%timeit avgs = poke_stats.mean(axis=1)
```

```
23.1 µs ± 235 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
%%timeit
avgs = []
for row in poke_stats:
    avg = np.mean(row)
    avgs.append(avg)
```

```
5.54 ms ± 224 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Let's practice!

WRITING EFFICIENT PYTHON CODE

Note: Remember, you want to avoid looping as much as possible when writing Python code. In cases where looping is unavoidable, be sure to check your loops for one-time calculations and holistic conversions to make them more efficient.

Writing better loops

WRITING EFFICIENT PYTHON CODE

Note: When writing a loop is unavoidable, be sure to analyze the loop and move any one-time calculations outside.

Logan Thomas
Scientific Software Technical Trainer, Enthought

datacamp

Lesson caveat

- Some of the following loops can be eliminated with techniques covered in previous lessons.
- Examples in this lesson are used for **demonstrative** purposes.

Warning: *For demonstration purposes only*

Writing better loops

- Understand what is being done with each loop iteration
- Move one-time calculations outside (above) the loop
- Use holistic conversions outside (below) the loop
- Anything that is done **once** should be outside the loop

Moving calculations above a loop

```
import numpy as np

names = ['Absol', 'Aron', 'Jynx', 'Natu', 'Onix']
attacks = np.array([130, 70, 50, 50, 45])
for pokemon,attack in zip(names, attacks):
    total_attack_avg = attacks.mean()
    if attack > total_attack_avg:
        print(
            "{}'s attack: {} > average: {}!"
            .format(pokemon, attack, total_attack_avg)
        )
```

```
Absol's attack: 130 > average: 69.0!
Aron's attack: 70 > average: 69.0!
```

4. Moving calculations above a loop

We have a list of Pokémon names and an array of each Pokémon's corresponding attack value. We'd like to print the names of each Pokémon with an attack value greater than the average of all attack values. To do this, we'll use a loop that iterates over each Pokémon and their attack value. For each iteration, the total attack average is calculated by finding the mean value of all attacks. Then, each Pokémon's attack value is evaluated to see if it exceeds the total average. Can you spot the inefficiency? The total_attack_avg variable is being created with each iteration of the loop. **But, this calculation doesn't change between iterations since it is an overall average. We only need to calculate this value once.**

```
import numpy as np

names = ['Absol', 'Aron', 'Jynx', 'Natu', 'Onix']
attacks = np.array([130, 70, 50, 50, 45])
# Calculate total average once (outside the loop)
total_attack_avg = attacks.mean()
for pokemon,attack in zip(names, attacks):

    if attack > total_attack_avg:
        print(
            "{}'s attack: {} > average: {}!"
            .format(pokemon, attack, total_attack_avg)
        )
```

```
Absol's attack: 130 > average: 69.0!
Aron's attack: 70 > average: 69.0!
```

5. Moving calculations above a loop
By moving this calculation outside (or above) the loop, we calculate the total attack average only once. We get the same output, but this is a more efficient approach.

Moving calculations above a loop

```
%%timeit
for pokemon,attack in zip(names, attacks):

    total_attack_avg = attacks.mean()

    if attack > total_attack_avg:
        print(
            "{}'s attack: {} > average: {}!"
            .format(pokemon, attack, total_attack_avg)
        )
```

```
74.9 µs ± 3.42 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

6. Moving calculations above a loop
Comparing runtimes, we see that keeping the total_attack_avg calculation within the loop takes roughly 75 microseconds.

Moving calculations above a loop

```
%%timeit
# Calculate total average once (outside the loop)
total_attack_avg = attacks.mean()

for pokemon,attack in zip(names, attacks):

    if attack > total_attack_avg:
        print(
            "{}'s attack: {} > average: {}!"
            .format(pokemon, attack, total_attack_avg)
        )
```

```
37.5 µs ± 281 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

7. Moving calculations above a loop
Moving the calculation takes about half the time.

Using holistic conversions

```
names = ['Pikachu', 'Squirtle', 'Articuno', ...]
legend_status = [False, False, True, ...]
generations = [1, 1, 1, ...]
poke_data = []
for poke_tuple in zip(names, legend_status, generations):
    poke_list = list(poke_tuple)
    poke_data.append(poke_list)
print(poke_data)
```

```
[['Pikachu', False, 1], ['Squirtle', False, 1], ['Articuno', True, 1], ...]
```

8. Using holistic conversions

Another way to make loops more efficient is to use holistic conversions outside (or below) the loop. We have three lists from our dataset of 720 Pokémon: a list of each Pokémon's name, a list corresponding to whether or not a Pokémon has a legendary status, and a list of each Pokémon's generation. We want to combine these objects so that each name, status, and generation is stored in an individual list. To do this, we'll use a loop that iterates over the output of the zip function. Remember, zip returns a collection of tuples, so we need to convert each tuple into a list since we want to create a list of lists as our output. Now, we append each individual poke_list to our poke_data output variable. By printing the result, we see our desired list of lists. However, converting each tuple to a list within the loop is not very efficient.

Using holistic conversions

```
names = ['Pikachu', 'Squirtle', 'Articuno', ...]
legend_status = [False, False, True, ...]
generations = [1, 1, 1, ...]
poke_data_tuples = []
for poke_tuple in zip(names, legend_status, generations):
    poke_data_tuples.append(poke_tuple)
poke_data = [*map(list, poke_data_tuples)]
print(poke_data)
```

9. Using holistic conversions
Instead, we should collect all of our poke_tuples together, and use the map function to convert each tuple to a list. The loop no longer converts tuples to lists with each iteration. Instead, we moved this tuple to list conversion outside (or below) the loop. That way, we convert data types all at once (or holistically) rather than converting in each iteration.

```
[['Pikachu', False, 1], ['Squirtle', False, 1], ['Articuno', True, 1], ...]
```

```
%%timeit
poke_data = []
for poke_tuple in zip(names, legend_status, generations):
    poke_list = list(poke_tuple)
    poke_data.append(poke_list)
```

```
261 µs ± 23.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
%%timeit
poke_data_tuples = []
for poke_tuple in zip(names, legend_status, generations):
    poke_data_tuples.append(poke_tuple)

poke_data = [*map(list, poke_data_tuples)]
```

```
224 µs ± 1.67 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

10. Using holistic conversions

Runtimes show that converting each tuple to a list outside of the loop is more efficient.

Time for some practice!

WRITING EFFICIENT PYTHON CODE