Welcome!

WRITING EFFICIENT PYTHON CODE

Logan Thomas

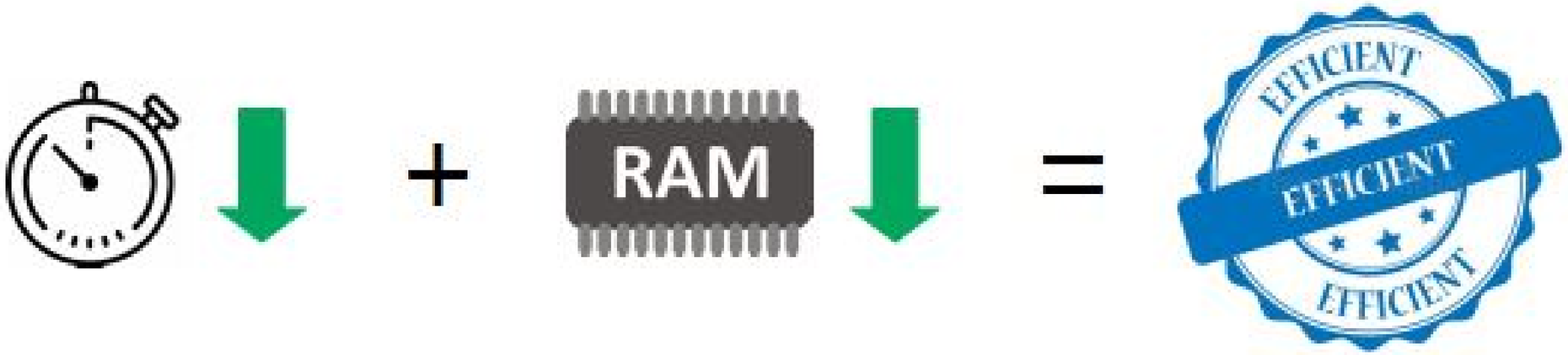
Senior Data Scientist, Protection Engineering Consultants

Course overview

- Your code should be a tool used to gain insights
  - Not something that leaves you waiting for results
- In this course, you will learn:
  - How to write clean, fast, and efficient Python code
  - How to profile your code for bottlenecks
  - How to eliminate bottlenecks and bad design patterns

Defining efficient

- Writing ***efficient*** Python code
  - Minimal completion time (*fast runtime*)
  - Minimal resource consumption (*small memory footprint*)



Defining Pythonic

- Writing efficient ***Python*** code
  - Focus on readability
  - Using Python's constructs as intended (i.e., *Pythonic*)

```
# Non-Pythonic
doubled_numbers = []

for i in range(len(numbers)):
    doubled_numbers.append(numbers[i] * 2)

# Pythonic
doubled_numbers = [x * 2 for x in numbers]
```

The Zen of Python by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
...
```

Things you should know

- Data types typically used in Data Science
  - **Data Types for Data Science**
- Writing and using your own functions
  - **Python Data Science Toolbox (Part 1)**
- Anonymous functions (`lambda` expressions)
  - **Python Data Science Toolbox (Part 1)**
- Writing and using list comprehensions
  - **Python Data Science Toolbox (Part 2)**

Let's get started!

WRITING EFFICIENT PYTHON CODE

Building with built-ins

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Senior Data Scientist, Protection Engineering Consultants

The Python Standard Library

- **Python 3.6 Standard Library**
  - Part of every standard Python installation
- Built-in types
  - `list` , `tuple` , `set` , `dict` , and others
- Built-in functions
  - `print()` , `len()` , `range()` , `round()` , `enumerate()` , `map()` , `zip()` , and others
- Built-in modules
  - `os` , `sys` , `itertools` , `collections` , `math` , and others

Built-in function: range()

Explicitly typing a list of numbers

```
nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Using `range()` to create the same list

```
# range(start,stop)
nums = range(0,11)

nums_list = list(nums)
print(nums_list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
# range(stop)
nums = range(11)

nums_list = list(nums)
print(nums_list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Built-in function: range()

Using `range()` with a step value

```
even_nums = range(2, 11, 2)

even_nums_list = list(even_nums)
print(even_nums_list)
```

```
[2, 4, 6, 8, 10]
```

Built-in function: enumerate()

Creates an indexed list of objects

```
letters = ['a', 'b', 'c', 'd' ]

indexed_letters = enumerate(letters)

indexed_letters_list = list(indexed_letters)
print(indexed_letters_list)
```

```
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

Built-in function: enumerate()

Can specify a start value

```
letters = ['a', 'b', 'c', 'd' ]

indexed_letters2 = enumerate(letters, start=5)

indexed_letters2_list = list(indexed_letters2)
print(indexed_letters2_list)
```

```
[(5, 'a'), (6, 'b'), (7, 'c'), (8, 'd')]
```

Built-in function: map()

Applies a function over an object

```
nums = [1.5, 2.3, 3.4, 4.6, 5.0]

rnd_nums = map(round, nums)

print(list(rnd_nums))
```

```
[2, 2, 3, 5, 5]
```

Built-in function: map()

`map()` with `lambda` (anonymous function)

```
nums = [1, 2, 3, 4, 5]

sqrd_nums = map(lambda x: x ** 2, nums)

print(list(sqrd_nums))
```

```
[1, 4, 9, 16, 25]
```

Let's start building with built-ins!

WRITING EFFICIENT PYTHON CODE

The power of NumPy arrays

WRITING EFFICIENT PYTHON CODE

Logan Thomas

Senior Data Scientist, Protection Engineering Consultants

NumPy array overview

- Alternative to Python lists

```
nums_list = list(range(5))
```

```
[0, 1, 2, 3, 4]
```

```
import numpy as np

nums_np = np.array(range(5))
```

```
array([0, 1, 2, 3, 4])
```

```
# NumPy array homogeneity
nums_np_ints = np.array([1, 2, 3])
```

```
array([1, 2, 3])
```

```
nums_np_ints.dtype
```

```
dtype('int64')
```

```
nums_np_floats = np.array([1, 2.5, 3])
```

```
array([1. , 2.5, 3. ])
```

```
nums_np_floats.dtype
```

```
dtype('float64')
```

NumPy array broadcasting

- Python lists don't support broadcasting

```
nums = [-2, -1, 0, 1, 2]
nums ** 2
```

```
TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

- List approach

```
# For loop (inefficient option)
sqrd_nums = []
for num in nums:
    sqrd_nums.append(num ** 2)
print(sqrd_nums)
```

```
[4, 1, 0, 1, 4]
```

```
# List comprehension (better option but not best)
sqrd_nums  = [num ** 2 for num in nums]

print(sqrd_nums)
```

```
[4, 1, 0, 1, 4]
```

NumPy array broadcasting

- NumPy array broadcasting for the win!

```
nums_np = np.array([-2, -1, 0, 1, 2])
nums_np ** 2
```

```
array([4, 1, 0, 1, 4])
```

Basic 1-D indexing (lists)

```
nums = [-2, -1, 0, 1, 2]
nums[2]
```

```
0
```

```
nums[-1]
```

```
2
```

```
nums[1:4]
```

```
[-1, 0, 1]
```

Basic 1-D indexing (arrays)

```
nums_np = np.array(nums)
nums_np[2]
```

```
0
```

```
nums_np[-1]
```

```
2
```

```
nums_np[1:4]
```

```
array([-1, 0, 1])
```

```
# 2-D list
nums2 = [ [1, 2, 3],
          [4, 5, 6] ]
```

```
# 2-D array

nums2_np = np.array(nums2)
```

- Basic 2-D indexing (lists)

```
nums2[0][1]
```

```
2
```

```
[row[0] for row in nums2]
```

```
[1, 4]
```

- Basic 2-D indexing (arrays)

```
nums2_np[0,1]
```

```
2
```

```
nums2_np[:,0]
```

```
array([1, 4])
```

NumPy array boolean indexing

```
nums = [-2, -1, 0, 1, 2]
nums_np =  np.array(nums)
```

- Boolean indexing

```
nums_np > 0
```

```
array([False, False, False,  True,  True])
```

```
nums_np[nums_np > 0]
```

```
array([1, 2])
```

- No boolean indexing for lists

```
# For loop (inefficient option)
pos = []
for num in nums:
    if num > 0:
        pos.append(num)
print(pos)
```

```
[1, 2]
```

```
# List comprehension (better option but not best)
pos = [num for num in nums if num > 0]
print(pos)
```

```
[1, 2]
```

Let's practice with powerful NumPy arrays!

WRITING EFFICIENT PYTHON CODE

Examining runtime

WRITING EFFICIENT PYTHON CODE

Logan Thomas

Senior Data Scientist, Protection Engineering Consultants

Why should we time our code?

- Allows us to pick the **optimal** coding approach
- Faster code == more efficient code!

How can we time our code?

- Calculate runtime with IPython magic command `%timeit`
- **Magic commands:** enhancements on top of normal Python syntax
  - Prefixed by the "%" character
  - Link to docs (**here**)
  - See all available magic commands with `%lsmagic`

Using %timeit

Code to be timed

```
import numpy as np

rand_nums = np.random.rand(1000)
```

Timing with `%timeit`

```
%timeit rand_nums = np.random.rand(1000)
```

```
8.61 µs ± 69.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

%timeit output

Code to be timed

```
rand_nums = np.random.rand(1000)
```

Timing with `%timeit`

```
%timeit rand_nums = np.random.rand(1000)
```

```
8.61 µs ± 69.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

%timeit output

Code to be timed

```
rand_nums = np.random.rand(1000)
```

Timing with `%timeit`

```
%timeit rand_nums = np.random.rand(1000)
```

```
8.61 µs ± 69.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

%timeit output

Code to be timed

```
rand_nums = np.random.rand(1000)
```

Timing with `%timeit`

```
%timeit rand_nums = np.random.rand(1000)
```

```
8.61 µs ± 69.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Specifying number of runs/loops

Setting the number of runs (`-r`) and/or loops (`-n`)

```
# Set number of runs to 2 (-r2)
# Set number of loops to 10 (-n10)

%timeit -r2 -n10 rand_nums = np.random.rand(1000)
```

```
16.9 µs ± 5.14 µs per loop (mean ± std. dev. of 2 runs, 10 loops each)
```

Using %timeit in line magic mode

Line magic (`%timeit`)

```
# Single line of code

%timeit nums = [x for x in range(10)]
```

```
914 ns ± 7.33 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Using %timeit in cell magic mode

Cell magic (`%%timeit`)

```
# Multiple lines of code


%%timeit
nums = []
for x in range(10):
    nums.append(x)
```

```
1.17 µs ± 3.26 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Saving output

Saving the output to a variable (`-o`)

```
times = %timeit -o rand_nums = np.random.rand(1000)
```

```
8.69 µs ± 91.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
times.timings
```

```
[8.697893059998023e-06,
 8.651204760008113e-06,
 8.634270530001232e-06,
 8.66847825998775e-06,
 8.619398139999247e-06,
 8.902550710008654e-06,
 8.633500570012985e-06]
```

```
times.best
```

```
8.619398139999247e-06
```

```
times.worst
```

```
8.902550710008654e-06
```

Comparing times

Python data structures can be created using formal name

```
formal_list = list()
formal_dict = dict()
formal_tuple = tuple()
```

Python data structures can be created using literal syntax

```
literal_list = []
literal_dict = {}
literal_tuple = ()
```

```
f_time = %timeit -o formal_dict = dict()
```

```
145 ns ± 1.5 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

```
l_time = %timeit -o literal_dict = {}
```

```
93.3 ns ± 1.88 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

```
diff = (f_time.average - l_time.average) * (10**9)
print('l_time better than f_time by {} ns'.format(diff))
```

```
l_time better than f_time by 51.90819192857814 ns
```

Comparing times

```
%timeit formal_dict = dict()
```

```
145 ns ± 1.5 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

```
%timeit literal_dict = {}
```

```
93.3 ns ± 1.88 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Off to the races!

WRITING EFFICIENT PYTHON CODE

Code profiling for runtime

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Senior Data Scientist, Protection Engineering Consultants

Code profiling

- Detailed stats on frequency and duration of function calls
- Line-by-line analyses
- Package used: `line_profiler`

```
pip install line_profiler
```

Code profiling: runtime

```
heroes = ['Batman', 'Superman', 'Wonder Woman']

hts = np.array([188.0, 191.0, 183.0])

wts = np.array([ 95.0, 101.0,  74.0])
```

```
def convert_units(heroes, heights, weights):

    new_hts = [ht * 0.39370  for ht in heights]
    new_wts = [wt * 2.20462  for wt in weights]

    hero_data = {}

    for i,hero in enumerate(heroes):
        hero_data[hero] = (new_hts[i], new_wts[i])

    return hero_data
```

```
convert_units(heroes, hts, wts)
```

```
{'Batman': (74.0156, 209.4389),
 'Superman': (75.1967, 222.6666),
 'Wonder Woman': (72.0471, 163.1419)}
```

Code profiling: runtime

```
%timeit convert_units(heroes, hts, wts)
```

```
3 µs ± 32 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
%timeit new_hts = [ht * 0.39370  for ht in hts]
```

```
1.09 µs ± 11 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
%timeit new_wts = [wt * 2.20462  for wt in wts]
```

```
1.08 µs ± 6.42 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
%%timeit
hero_data = {}
for i,hero in enumerate(heroes):
    hero_data[hero] = (new_hts[i], new_wts[i])
```

```
634 ns ± 9.29 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Code profiling: line_profiler

Using `line_profiler` package

```
%load_ext line_profiler
```

Magic command for line-by-line times

```
%lprun -f
```

Code profiling: line_profiler

Using `line_profiler` package

```
%load_ext line_profiler
```

Magic command for line-by-line times

```
%lprun -f convert_units
```

Code profiling: line_profiler

Using `line_profiler` package

```
%load_ext line_profiler
```

Magic command for line-by-line times

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

%lprun output

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

```
%timeit convert_units convert_units(heroes, hts, wts)
```

```
3 µs ± 32 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
%lprun -f convert_units convert_units(heroes, hts, wts)
```

```
Timer unit: 1e-06 s

Total time: 2.6e-05 s
File: <ipython-input-211-2e40813f07a3>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1         13.0     13.0     50.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1          4.0      4.0     15.4      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      3.8      hero_data = {}
     7
     8         4          4.0      1.0     15.4      for i,hero in enumerate(heroes):
     9         3          3.0      1.0     11.5          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      3.8      return hero_data
```

Let's practice your new profiling powers!

WRITING EFFICIENT PYTHON CODE

Code profiling for memory usage

WRITING EFFICIENT PYTHON CODE

Logan Thomas

Senior Data Scientist, Protection Engineering Consultants

Quick and dirty approach

```
import sys
```

```
nums_list = [*range(1000)]
sys.getsizeof(nums_list)
```

```
9112
```

```
import numpy as np

nums_np = np.array(range(1000))
sys.getsizeof(nums_np)
```

```
8096
```

Code profiling: memory

- Detailed stats on memory consumption
- Line-by-line analyses
- Package used: `memory_profiler`

```
pip install memory_profiler
```

  - Using `memory_profiler` package

```
%load_ext memory_profiler


%mprun -f convert_units convert_units(heroes, hts, wts)
```

Code profiling: memory

- Functions must be imported when using `memory_profiler`
  - `hero_funcs.py`

```
from hero_funcs import convert_units
```

```
%load_ext memory_profiler


%mprun -f convert_units convert_units(heroes, hts, wts)
```

%mprun output

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

%mprun output

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

%mprun output

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

%mprun output

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

%mprun output

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

%mprun output

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

%mprun output caveats

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

%mprun output caveats

Data used in this example is a random sample of 35,000 heroes.

(not original 480 superheroes dataset)

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1    103.8 MiB    103.8 MiB   def convert_units(heroes, heights, weights):
     2
     3    103.9 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4    104.1 MiB      0.2 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6    104.1 MiB      0.0 MiB       hero_data = {}
     7
     8    104.3 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9    104.3 MiB      0.2 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11    104.3 MiB      0.0 MiB       return hero_data
```

%mprun output caveats

Small memory allocations could result in `0.0 MiB` output.

(using original 480 superheroes dataset)

```
%mprun -f convert_units convert_units(heroes, hts, wts)
```

```
Filename: ~/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     1     98.7 MiB     98.7 MiB   def convert_units(heroes, heights, weights):
     2
     3     98.7 MiB      0.0 MiB       new_hts = [ht * 0.39370  for ht in heights]
     4     98.7 MiB      0.0 MiB       new_wts = [wt * 2.20462  for wt in weights]
     5
     6     98.7 MiB      0.0 MiB       hero_data = {}
     7
     8     98.7 MiB      0.0 MiB       for i,hero in enumerate(heroes):
     9     98.7 MiB      0.0 MiB           hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11     98.7 MiB      0.0 MiB       return hero_data
```

%mprun output caveats

- Inspects memory by querying the operating system
- Results may differ between platforms and runs
  - Can still observe how each line of code compares to others based on memory consumption

Let's practice!

WRITING EFFICIENT PYTHON CODE

Efficiently combining, counting, and iterating

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Senior Data Scientist, Protection Engineering Consultants

Pokémon Overview

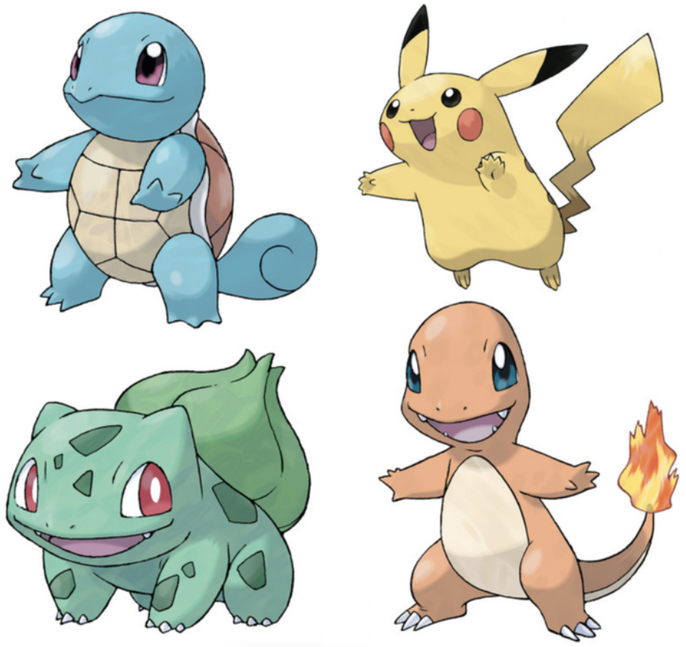
- Trainers (collect Pokémon)

`Trainer`

Pokémon Overview

- Pokémon (fictional animal characters)

Trainer

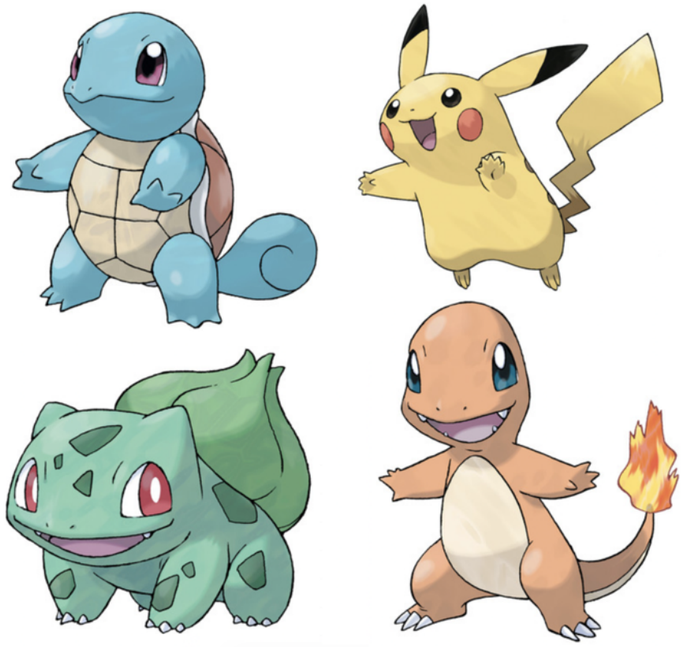


Pokémon

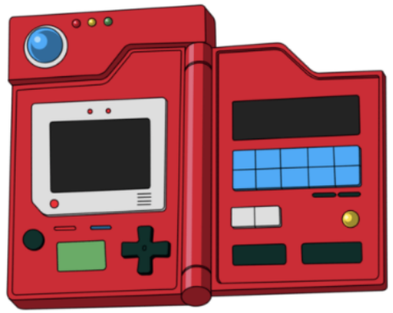
Pokémon Overview

- 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)]
```

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

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)
```

```
{'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)
```

```
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`

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`**

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)
```

```
[('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')]
```

Let's practice!

WRITING EFFICIENT PYTHON CODE

Set theory

WRITING EFFICIENT PYTHON CODE

Logan Thomas

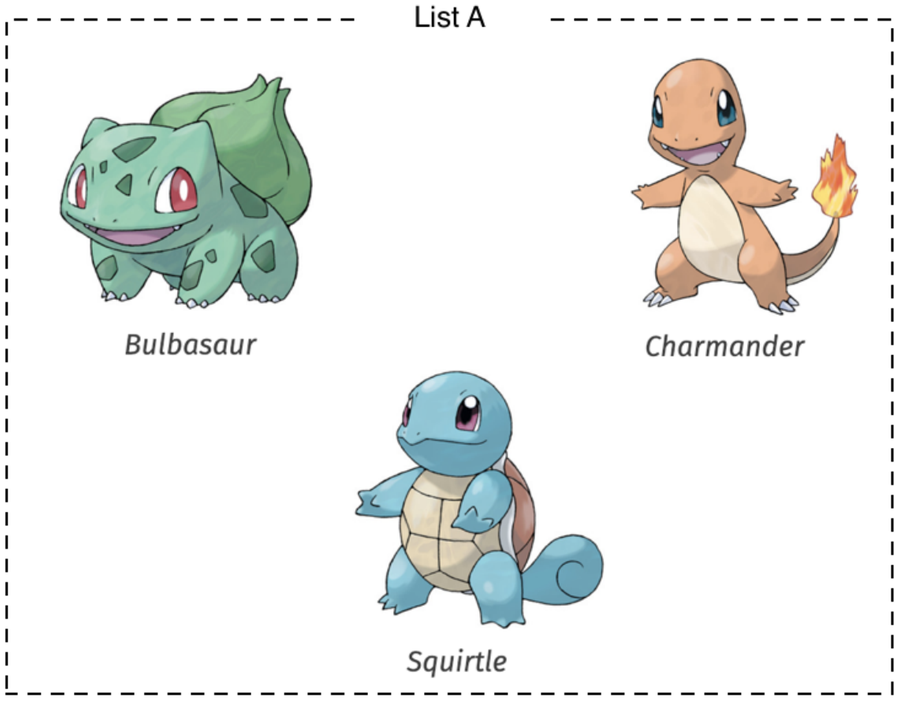
Senior Data Scientist, Protection Engineering Consultants

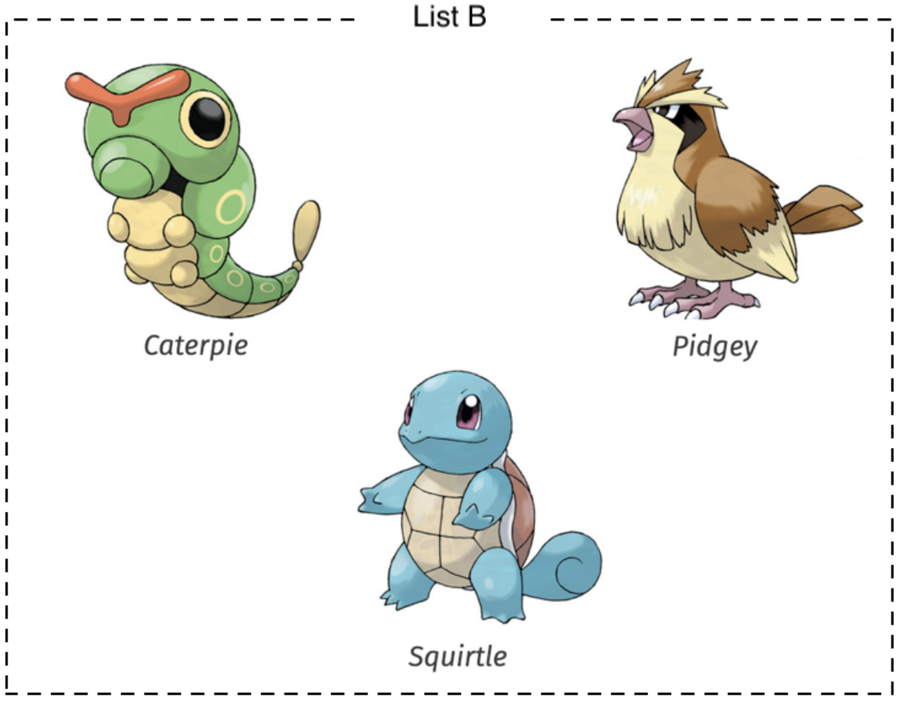
Set theory

- 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

Comparing objects with loops

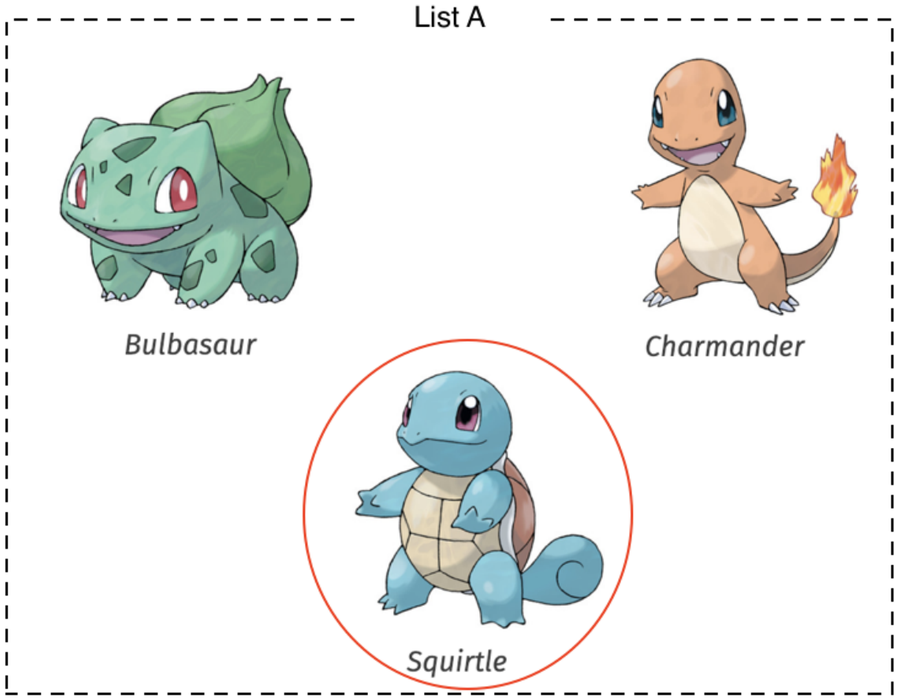
```
list_a = ['Bulbasaur', 'Charmander', 'Squirtle']
list_b = ['Caterpie', 'Pidgey', 'Squirtle']
```

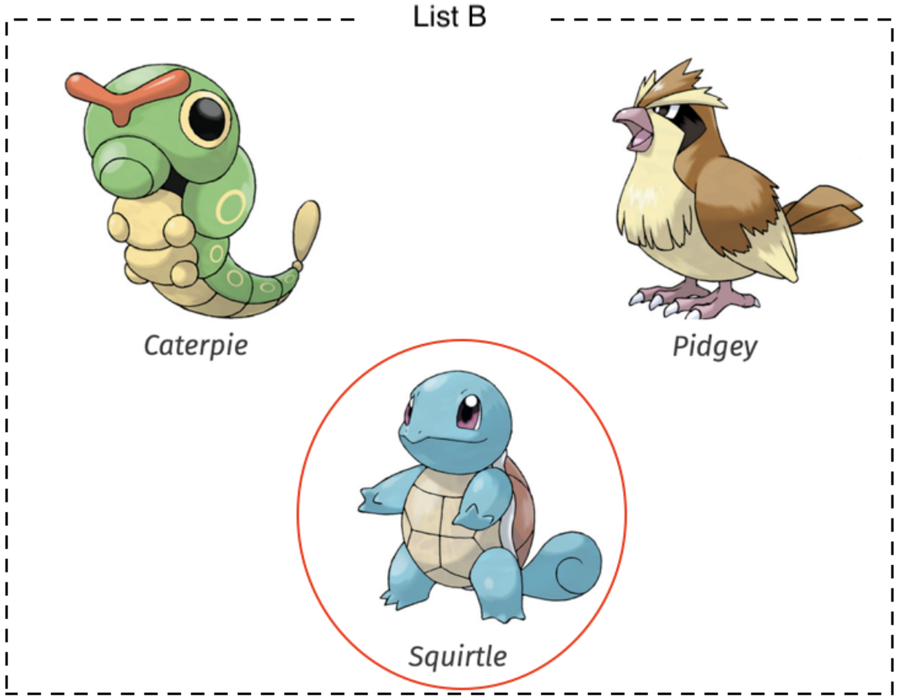




Comparing objects with loops

```
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)
```

```
['Squirtle']
```

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

```
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)
```

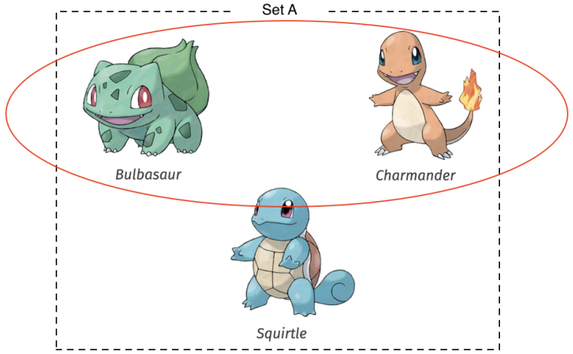
```
137 ns ± 3.01 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

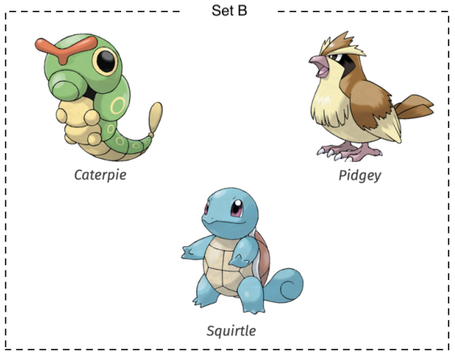
Set method: difference

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

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

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



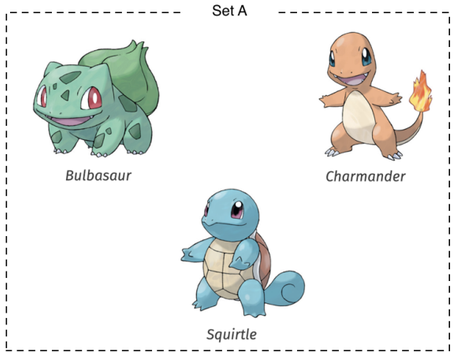


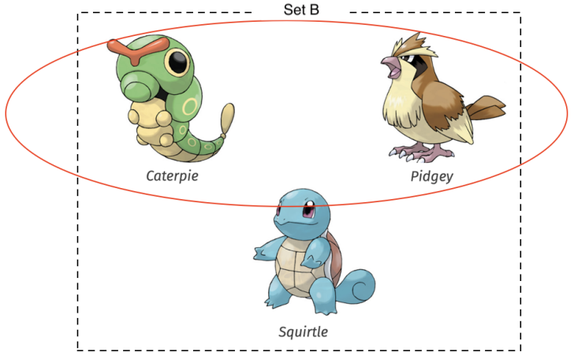
Set method: difference

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

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

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



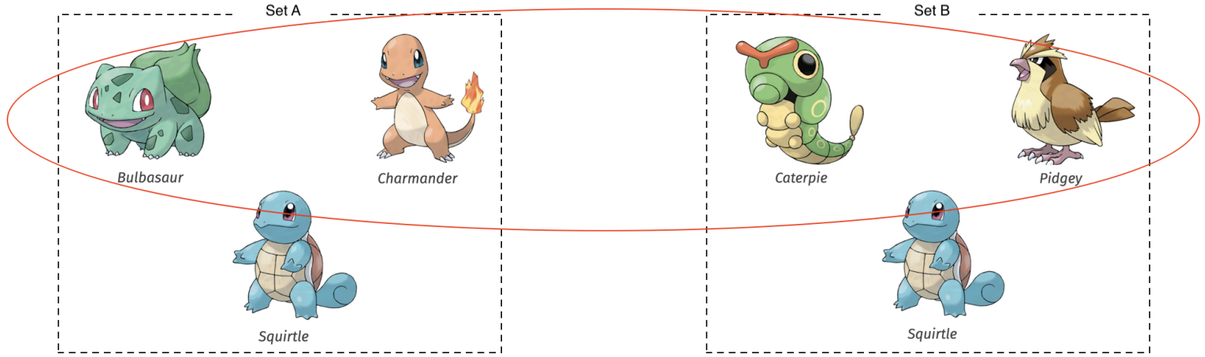


Set method: symmetric difference

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

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

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

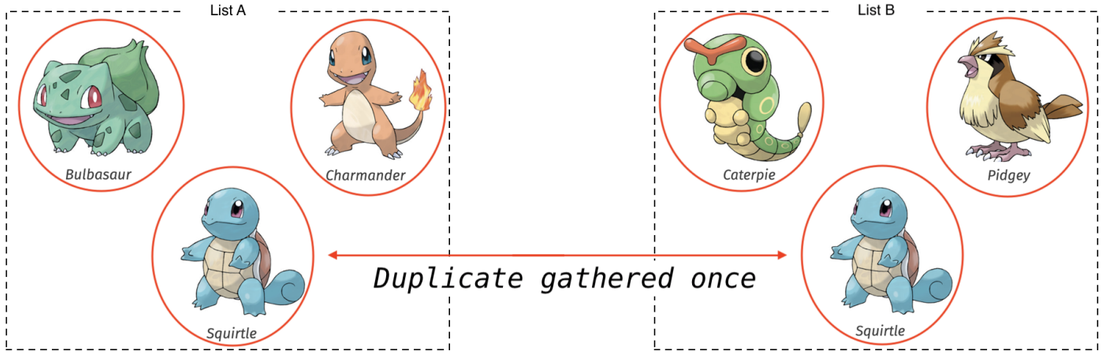


Set method: union

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

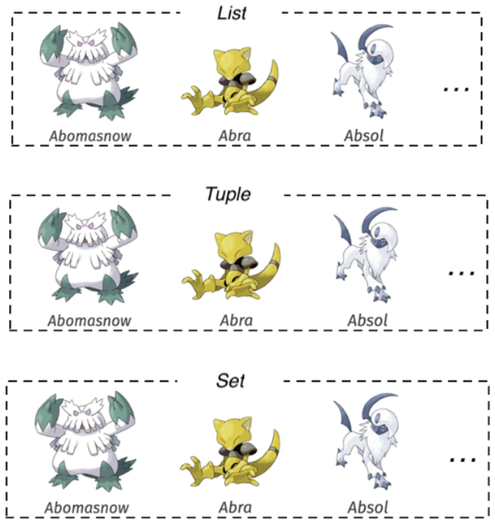
```
set_a.union(set_b)
```

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



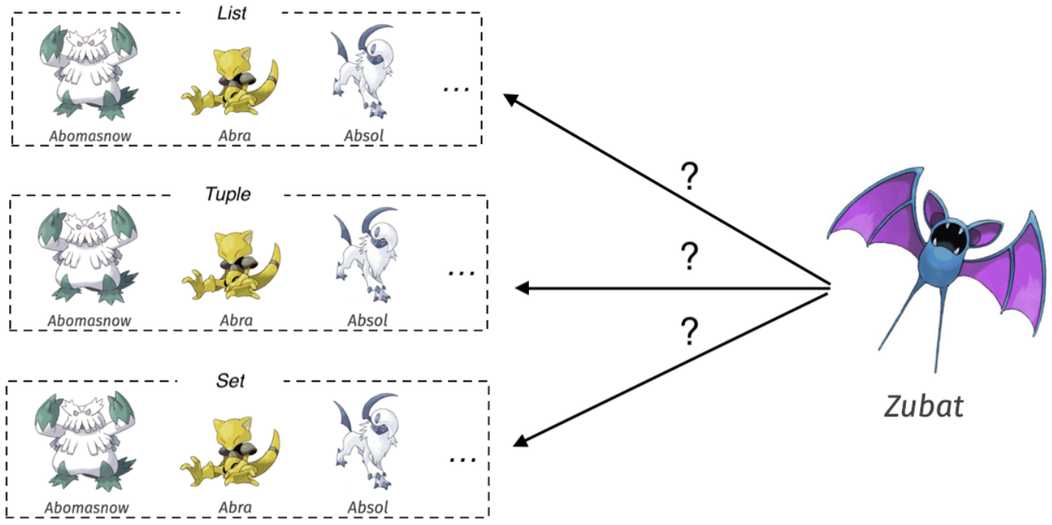
Membership testing with sets

```
# 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)
```

```
['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)
```

```
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

Senior Data Scientist, Protection Engineering Consultants

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)]
```

```
%%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, ...]
```

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

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

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

Writing better loops

WRITING EFFICIENT PYTHON CODE

Logan Thomas

Senior Data Scientist, Protection Engineering Consultants

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!
```

```
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!
```

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)
```

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)
```

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], ...]
```

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)
```

```
[['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)
```

Time for some practice!

WRITING EFFICIENT PYTHON CODE

Intro to pandas DataFrame iteration

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Senior Data Scientist, Protection Engineering Consultants

pandas recap

- See pandas overview in **Intermediate Python for Data Science**
- Library used for data analysis
- Main data structure is the DataFrame
  - Tabular data with labeled rows and columns
  - Built on top of the NumPy array structure
- Chapter Objective:
  - Best practice for iterating over a pandas DataFrame

Baseball stats

```
import pandas as pd

baseball_df = pd.read_csv('baseball_stats.csv')
print(baseball_df.head())
```

```
  Team League  Year   RS   RA   W    G  Playoffs
0  ARI     NL  2012  734  688  81  162         0
1  ATL     NL  2012  700  600  94  162         1
2  BAL     AL  2012  712  705  93  162         1
3  BOS     AL  2012  734  806  69  162         0
4  CHC     NL  2012  613  759  61  162         0
```

Baseball stats

```
   Team
0   ARI
1   ATL
2   BAL
3   BOS
4   CHC
```

Arizona Diamondbacks (ARI)

Atlanta Braves (ATL)

Baltimore Orioles (BAL)

Boston Red Sox (BOS)

Chicago Cubs (CHC)

Baseball stats

```
   Team League  Year   RS   RA   W    G  Playoffs
0   ARI     NL  2012  734  688  81  162         0
1   ATL     NL  2012  700  600  94  162         1
2   BAL     AL  2012  712  705  93  162         1
3   BOS     AL  2012  734  806  69  162         0
4   CHC     NL  2012  613  759  61  162         0
```

Calculating win percentage

```
import numpy as np

def calc_win_perc(wins, games_played):

    win_perc = wins / games_played

    return np.round(win_perc,2)
```

```
win_perc = calc_win_perc(50, 100)
print(win_perc)
```

```
0.5
```

Adding win percentage to DataFrame

```
win_perc_list = []

for i in range(len(baseball_df)):
    row = baseball_df.iloc[i]

    wins = row['W']
    games_played = row['G']

    win_perc = calc_win_perc(wins, games_played)

    win_perc_list.append(win_perc)

baseball_df['WP'] = win_perc_list
```

Adding win percentage to DataFrame

```
print(baseball_df.head())
```

```
  Team League  Year   RS   RA   W    G  Playoffs    WP
0  ARI     NL  2012  734  688  81  162         0  0.50
1  ATL     NL  2012  700  600  94  162         1  0.58
2  BAL     AL  2012  712  705  93  162         1  0.57
3  BOS     AL  2012  734  806  69  162         0  0.43
4  CHC     NL  2012  613  759  61  162         0  0.38
```

Iterating with .iloc

```
%%timeit
win_perc_list = []

for i in range(len(baseball_df)):
    row = baseball_df.iloc[i]

    wins = row['W']
    games_played = row['G']

    win_perc = calc_win_perc(wins, games_played)
    win_perc_list.append(win_perc)

baseball_df['WP'] = win_perc_list
```

```
183 ms ± 1.73 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Iterating with .iterrows()

```
win_perc_list = []

for i,row in baseball_df.iterrows():
    wins = row['W']
    games_played = row['G']

    win_perc = calc_win_perc(wins, games_played)

    win_perc_list.append(win_perc)

baseball_df['WP'] = win_perc_list
```

Iterating with .iterrows()

```
%%timeit
win_perc_list = []

for i,row in baseball_df.iterrows():

    wins = row['W']
    games_played = row['G']

    win_perc = calc_win_perc(wins, games_played)
    win_perc_list.append(win_perc)

baseball_df['WP'] = win_perc_list
```

```
95.3 ms ± 3.57 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Practice DataFrame iterating with .iterrows()

WRITING EFFICIENT PYTHON CODE

Another iterator method: .itertuples()

WRITING EFFICIENT PYTHON CODE

Logan Thomas

Senior Data Scientist, Protection Engineering Consultants

Team wins data

```
print(team_wins_df)
```

```
     Team  Year    W
0     ARI  2012   81
1     ATL  2012   94
2     BAL  2012   93
3     BOS  2012   69
4     CHC  2012   61
...
```

```
for row_tuple in team_wins_df.iterrows():
    print(row_tuple)
    print(type(row_tuple[1]))
```

```
(0, Team     ARI
Year    2012
W         81
Name: 0, dtype: object)
<class 'pandas.core.series.Series'>

(1, Team     ATL
Year    2012
W         94
Name: 1, dtype: object)
<class 'pandas.core.series.Series'>
...
```

Iterating with .itertuples()

```
for row_namedtuple in team_wins_df.itertuples():
    print(row_namedtuple)
```

```
Pandas(Index=0, Team='ARI', Year=2012, W=81)
Pandas(Index=1, Team='ATL', Year=2012, W=94)
...
```

```
print(row_namedtuple.Index)
```

```
1
```

```
print(row_namedtuple.Team)
```

```
ATL
```

Comparing methods

```
%%timeit
for row_tuple in team_wins_df.iterrows():
    print(row_tuple)
```

```
527 ms ± 41.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
for row_namedtuple in team_wins_df.itertuples():
    print(row_namedtuple)
```

```
7.48 ms ± 243 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
for row_tuple in team_wins_df.iterrows():
    print(row_tuple[1]['Team'])
```

```
ARI
ATL
...
```

```
for row_namedtuple in team_wins_df.itertuples():
    print(row_namedtuple['Team'])
```

```
TypeError: tuple indices must be integers or slices, not str
```

```
for row_namedtuple in team_wins_df.itertuples():
    print(row_namedtuple.Team)
```

```
ARI
ATL
...
```

Let's keep iterating!

WRITING EFFICIENT PYTHON CODE

pandas alternative to looping

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Senior Data Scientist, Protection Engineering Consultants

```
print(baseball_df.head())
```

```
  Team League  Year   RS   RA   W    G  Playoffs
0  ARI     NL  2012  734  688  81  162         0
1  ATL     NL  2012  700  600  94  162         1
2  BAL     AL  2012  712  705  93  162         1
3  BOS     AL  2012  734  806  69  162         0
4  CHC     NL  2012  613  759  61  162         0
```

```
def calc_run_diff(runs_scored, runs_allowed):

    run_diff = runs_scored - runs_allowed

    return run_diff
```

Run differentials with a loop

```
run_diffs_iterrows = []

for i,row in baseball_df.iterrows():
    run_diff = calc_run_diff(row['RS'], row['RA'])
    run_diffs_iterrows.append(run_diff)

baseball_df['RD'] = run_diffs_iterrows
print(baseball_df)
```

```
     Team League  Year    RS   RA     W    G  Playoffs   RD
0     ARI     NL  2012   734  688    81  162         0   46
1     ATL     NL  2012   700  600    94  162         1  100
2     BAL     AL  2012   712  705    93  162         1    7
...
```

pandas .apply() method

- Takes a function and applies it to a DataFrame
  - Must specify an axis to apply (`0` for columns; `1` for rows)
- Can be used with anonymous functions (`lambda` functions)
- Example:

```
baseball_df.apply(
    lambda row: calc_run_diff(row['RS'], row['RA']),
    axis=1
)
```

Run differentials with .apply()

```
run_diffs_apply = baseball_df.apply(
          lambda row: calc_run_diff(row['RS'], row['RA']),
          axis=1)

baseball_df['RD'] = run_diffs_apply
print(baseball_df)
```

```
     Team League  Year    RS    RA     W    G  Playoffs    RD
0     ARI     NL  2012   734   688    81   162          0    46
1     ATL     NL  2012   700   600    94   162          1   100
2     BAL     AL  2012   712   705    93   162          1     7
...
```

Comparing approaches

```
%%timeit
run_diffs_iterrows = []

for i,row in baseball_df.iterrows():
    run_diff = calc_run_diff(row['RS'], row['RA'])
    run_diffs_iterrows.append(run_diff)

baseball_df['RD'] = run_diffs_iterrows
```

```
86.8 ms ± 3 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Comparing approaches

```
%%timeit
run_diffs_apply = baseball_df.apply(
        lambda row: calc_run_diff(row['RS'], row['RA']),
        axis=1)

baseball_df['RD'] = run_diffs_apply
```

```
30.1 ms ± 1.75 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Let's practice using pandas .apply() method!

WRITING EFFICIENT PYTHON CODE

Optimal pandas iterating

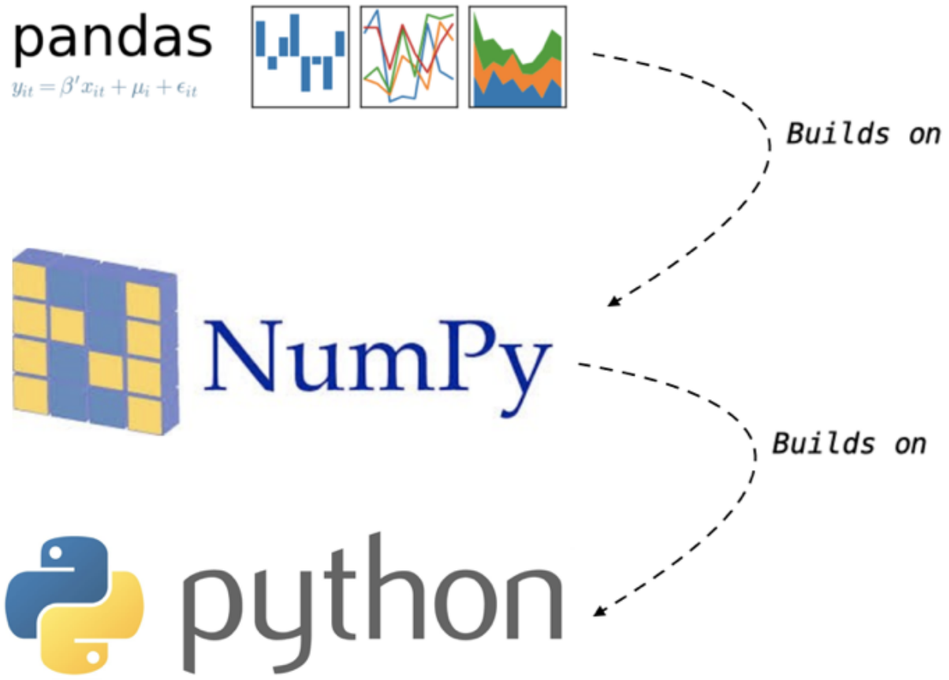
WRITING EFFICIENT PYTHON CODE

Logan Thomas

Senior Data Scientist, Protection Engineering Consultants

pandas internals

- Eliminating loops applies to using pandas as well
- pandas is built on NumPy
  - Take advantage of NumPy array efficiencies



```
print(baseball_df)
```

```
  Team League  Year   RS   RA   W    G  Playoffs
0  ARI     NL  2012  734  688  81  162         0
1  ATL     NL  2012  700  600  94  162         1
2  BAL     AL  2012  712  705  93  162         1
...
```

```
wins_np = baseball_df['W'].values
print(type(wins_np))
```

```
<class 'numpy.ndarray'>
```

```
print(wins_np)
```

```
[ 81  94  93 ...]
```

Power of vectorization

- Broadcasting (vectorizing) is extremely efficient!

```
baseball_df['RS'].values - baseball_df['RA'].values
```

```
array([  46,  100,    7, ...,  188,  110, -117])
```

Run differentials with arrays

```
run_diffs_np = baseball_df['RS'].values - baseball_df['RA'].values

baseball_df['RD'] = run_diffs_np
print(baseball_df)
```

```
     Team League  Year   RS   RA    W    G  Playoffs   RD
0     ARI     NL  2012  734  688   81  162         0   46
1     ATL     NL  2012  700  600   94  162         1  100
2     BAL     AL  2012  712  705   93  162         1    7
3     BOS     AL  2012  734  806   69  162         0  -72
4     CHC     NL  2012  613  759   61  162         0 -146
...
```

Comparing approaches

```
%%timeit
run_diffs_np = baseball_df['RS'].values - baseball_df['RA'].values

baseball_df['RD'] = run_diffs_np
```

```
124 µs ± 1.47 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Let's put our skills into practice!

WRITING EFFICIENT PYTHON CODE

Congratulations!

WRITING EFFICIENT PYTHON CODE

Logan Thomas

Senior Data Scientist, Protection Engineering Consultants

What you have learned

- The definition of **efficient** and **Pythonic** code
- How to use Python's powerful built-in library
- The advantages of NumPy arrays
- Some handy magic commands to profile code
- How to deploy efficient solutions with `zip()`, `itertools`, `collections`, and set theory
- The cost of looping and how to eliminate loops
- Best practices for iterating with pandas DataFrames

Well done!

WRITING EFFICIENT PYTHON CODE