```
---
title: Writing Efficient Python Code
tags: python
url: https://www.datacamp.com/courses/writing-efficient-python-code
---

# 1. Foundations for efficiencies
## A taste of things to come
```python
# Print the list created using the Non-Pythonic approach
i = 0
new_list= []
while i < len(names):
    if len(names[i]) >= 6:
        new_list.append(names[i])
    i += 1
print(new_list)
```
```python
# Print the list created by using list comprehension
best_list = [name for name in names if len(name) >= 6]
print(best_list)
```

## Zen of Python
```python
# Create a range object that goes from 0 to 5
nums = range(6)
print(type(nums))

# Convert nums to a list
nums_list = list(nums)
print(nums_list)

# Create a new list of odd numbers from 1 to 11 by unpacking a range object
nums_list2 = [*range(1, 12, 2)]
print(nums_list2)
```

## Buit-in practice: enumerate()
```python
# Rewrite the for loop to use enumerate
indexed_names = []
for i, name in enumerate(names):
    index_name = (i,name)
    indexed_names.append(index_name)
print(indexed_names)

# Rewrite the above for loop using list comprehension
indexed_names_comp = [(i, name) for i,name in enumerate(names)]
print(indexed_names_comp)

# Unpack an enumerate object with a starting index of one
indexed_names_unpack = [*enumerate(names, start=1)]
print(indexed_names_unpack)
```

## Built-in practice: map()
```python
# Use map to apply str.upper to each element in names
names_map  = map(lambda name: name.upper(), names)

# Print the type of the names_map
print(type(names_map))
```
```

```python
# Unpack names_map into a list
names_uppercase = [*names_map]

# Print the list created above
print(names_uppercase)
```

## Practice with NumPy arrays
```python
# Print second row of nums
print(nums[1, :])

# Print all elements of nums that are greater than six
print(nums[nums > 6])

# Double every element of nums
nums_dbl = nums * 2
print(nums_dbl)

# Replace the third column of nums
nums[:,2] = nums[:,2] + 1
print(nums)
```

## Bringing it all together: Festivus!
```python
# Create a list of arrival times
arrival_times = [*range(10,60,10)]

# Convert arrival_times to an array and update the times
arrival_times_np = np.array(arrival_times)
new_times = arrival_times_np - 3

# Use list comprehension and enumerate to pair guests to new times
guest_arrivals = [(names[i],time) for i,time in enumerate(new_times)]

# Map the welcome_guest function to each (guest,time) pair
welcome_map = map(welcome_guest, guest_arrivals)

guest_welcomes = [*welcome_map]
print(*guest_welcomes, sep='\n')
```

# 2. Timing and profiling code
## Using %timeit: your turn!
```python
In [1]: %timeit nums_list_comp = [num for num in range(51)]
3.05 us +- 436 ns per loop (mean +- std. dev. of 7 runs, 1000000 loops each)

In [3]: %timeit nums_unpack = [*range(51)]
488 ns +- 29.6 ns per loop (mean +- std. dev. of 7 runs, 1000000 loops each)
```

## Using %timeit: specifying number of runs and loops
```python
%%timeit -r5 -n25 set(heroes)
10.1 us +- 1.45 us per loop (mean +- std. dev. of 5 runs, 25 loops each)
```

## Using %timeit: formal name or literal syntax
```python
In [2]: %timeit literal_list = []
20.7 ns +- 1.52 ns per loop (mean +- std. dev. of 7 runs, 10000000 loops each)
```

```
In [3]: %timeit formal_list = list()
84.1 ns +- 6.77 ns per loop (mean +- std. dev. of 7 runs, 10000000 loops each)
```

## Using cell magic mode (%%timeit)

```python
In [2]: %%timeit hero_wts_lbs = []
... for wt in wts:
...     hero_wts_lbs.append(wt * 2.20462)
1.23 ms +- 212 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)

In [3]: %%timeit wts_np = np.array(wts)
... hero_wts_lbs_np = wts_np * 2.20462
1.12 us +- 26.8 ns per loop (mean +- std. dev. of 7 runs, 1000000 loops each)
```

## Using %lprun: spot bottlenecks

```python
In [1]: %load_ext line_profiler
In [3]: %lprun -f convert_units convert_units(heroes, hts, wts)
Timer unit: 1e-06 s

Total time: 0.000916 s
File: <ipython-input-1-2ae8c0194a47>
Function: convert_units at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units(heroes, heights, weights):
     2
     3         1        128.0    128.0     14.0      new_hts = [ht * 0.39370  for ht in heights]
     4         1        109.0    109.0     11.9      new_wts = [wt * 2.20462  for wt in weights]
     5
     6         1          1.0      1.0      0.1      hero_data = {}
     7
     8       481        330.0      0.7     36.0      for i,hero in enumerate(heroes):
     9       480        347.0      0.7     37.9          hero_data[hero] = (new_hts[i], new_wts[i])
    10
    11         1          1.0      1.0      0.1      return hero_data
```

## Using %lprun: fix the bottleneck

```python
In [2]: %lprun -f convert_units_broadcast convert_units_broadcast(heroes, hts, wts)
Timer unit: 1e-06 s

Total time: 0.000585 s
File: <ipython-input-3-84e44a6b12f5>
Function: convert_units_broadcast at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def convert_units_broadcast(heroes, heights,
weights):
     2
     3                                               # Array broadcasting instead of list
comprehension
     4         1         30.0     30.0      5.1      new_hts = heights * 0.39370
     5         1          3.0      3.0      0.5      new_wts = weights * 2.20462
     6
     7         1          1.0      1.0      0.2      hero_data = {}
     8
     9       481        231.0      0.5     39.5      for i,hero in enumerate(heroes):
    10       480        319.0      0.7     54.5          hero_data[hero] = (new_hts[i], new_wts[i])
    11
```

```
    12         1          1.0      1.0      0.2      return hero_data
```

## Code profiling for memory usage
```python
In [1]: %load_ext memory_profiler
In [2]: from bmi_lists import calc_bmi_lists
In [3]: %mprun -f calc_bmi_lists calc_bmi_lists(sample_indices, hts, wts)
Filename: /tmp/tmp00h_xy37/bmi_lists.py

Line #    Mem usage    Increment   Line Contents
================================================
     1     79.3 MiB     79.3 MiB   def calc_bmi_lists(sample_indices, hts, wts):
     2
     3                                 # Gather sample heights and weights as lists
     4     79.9 MiB      0.3 MiB       s_hts = [hts[i] for i in sample_indices]
     5     80.8 MiB      0.3 MiB       s_wts = [wts[i] for i in sample_indices]
     6
     7                                 # Convert heights from cm to m and square with list
comprehension
     8     81.7 MiB      0.4 MiB       s_hts_m_sqr = [(ht / 100) ** 2 for ht in s_hts]
     9
    10                                 # Calculate BMIs as a list with list comprehension
    11     82.3 MiB      0.3 MiB       bmis = [s_wts[i] / s_hts_m_sqr[i] for i in
range(len(sample_indices))]
    12
    13     82.3 MiB      0.0 MiB       return bmis
```

## Using %mprun: Hero BMI 2.0
```python
In [1]: %load_ext memory_profiler
The memory_profiler extension is already loaded. To reload it, use:
  %reload_ext memory_profiler

In [2]: from bmi_arrays import calc_bmi_arrays

In [3]: %mprun -f calc_bmi_arrays calc_bmi_arrays(sample_indices, hts, wts)
Filename: /tmp/tmpu3bg1r51/bmi_arrays.py

Line #    Mem usage    Increment   Line Contents
================================================
     1     78.7 MiB     78.7 MiB   def calc_bmi_arrays(sample_indices, hts, wts):
     2
     3                                 # Gather sample heights and weights as arrays
     4     78.7 MiB      0.1 MiB       s_hts = hts[sample_indices]
     5     79.0 MiB      0.3 MiB       s_wts = wts[sample_indices]
     6
     7                                 # Convert heights from cm to m and square with broadcasting
     8     79.3 MiB      0.3 MiB       s_hts_m_sqr = (s_hts / 100) ** 2
     9
    10                                 # Calculate BMIs as an array using broadcasting
    11     79.3 MiB      0.0 MiB       bmis = s_wts / s_hts_m_sqr
    12
    13     79.3 MiB      0.0 MiB       return bmis
```

## Bringing it all together: Star Wars profiling
```python
In [1]: %load_ext line_profiler

In [2]: %lprun -f get_publisher_heroes get_publisher_heroes(heroes, publishers, "George Lucas")
Timer unit: 1e-06 s

Total time: 0.000301 s
```

```
File: <ipython-input-2-5a6bc05c1c55>
Function: get_publisher_heroes at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def get_publisher_heroes(heroes, publishers,
desired_publisher):
     2
     3         1          2.0      2.0      0.7      desired_heroes = []
     4
     5       481        147.0      0.3     48.8      for i,pub in enumerate(publishers):
     6       480        140.0      0.3     46.5          if pub == desired_publisher:
     7         4         12.0      3.0      4.0              desired_heroes.append(heroes[i])
     8
     9         1          0.0      0.0      0.0      return desired_heroes

In [3]: %lprun -f get_publisher_heroes_np get_publisher_heroes_np(heroes, publishers, "George Lucas")
Timer unit: 1e-06 s

Total time: 0.000215 s
File: <ipython-input-2-5a6bc05c1c55>
Function: get_publisher_heroes_np at line 12

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    12                                           def get_publisher_heroes_np(heroes, publishers,
desired_publisher):
    13
    14         1        143.0    143.0     66.5      heroes_np = np.array(heroes)
    15         1         42.0     42.0     19.5      pubs_np = np.array(publishers)
    16
    17         1         29.0     29.0     13.5      desired_heroes = heroes_np[pubs_np ==
desired_publisher]
    18
    19         1          1.0      1.0      0.5      return desired_heroes

In [7]: from hero_funcs import get_publisher_heroes
In [8]: from hero_funcs import get_publisher_heroes_np
In [9]: %mprun -f get_publisher_heroes get_publisher_heroes(heroes, publishers, "George Lucas")
Filename: /tmp/tmpjjwz8b32/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
     4    110.4 MiB    110.4 MiB   def get_publisher_heroes(heroes, publishers, desired_publisher):
     5
     6    110.4 MiB      0.0 MiB       desired_heroes = []
     7
     8    110.4 MiB      0.0 MiB       for i,pub in enumerate(publishers):
     9    110.4 MiB      0.0 MiB           if pub == desired_publisher:
    10    110.4 MiB      0.0 MiB               desired_heroes.append(heroes[i])
    11
    12    110.4 MiB      0.0 MiB       return desired_heroes

In [1]: %mprun -f get_publisher_heroes_np get_publisher_heroes_np(heroes, publishers, "George Lucas")
Filename: /tmp/tmpjjwz8b32/hero_funcs.py

Line #    Mem usage    Increment   Line Contents
================================================
    15    110.4 MiB    110.4 MiB   def get_publisher_heroes_np(heroes, publishers,
desired_publisher):
    16
    17    110.4 MiB      0.0 MiB       heroes_np = np.array(heroes)
    18    110.4 MiB      0.0 MiB       pubs_np = np.array(publishers)
    19
    20    110.4 MiB      0.0 MiB       desired_heroes = heroes_np[pubs_np == desired_publisher]
```

```
    21
    22    110.4 MiB      0.0 MiB       return desired_heroes
```

# 3. Gaining efficiencies
## Combining Pokemon names and types
```python
# Combine five items from names and three items from primary_types
differing_lengths = [*zip(names[:5], primary_types[:3])]

print(*differing_lengths, sep='\n')
```

## Counting Pokemon from a sample
```python
# Collect the count of primary types
type_count = Counter(primary_types)
print(type_count, '\n')

# Collect the count of generations
gen_count = Counter(generations)
print(gen_count, '\n')

# Use list comprehension to get each Pok�mon's starting letter
starting_letters = [name[:1] for name in names]

# Collect the count of Pok�mon for each starting_letter
starting_letters_count = Counter(starting_letters)
print(starting_letters_count)
```

## Combinations of Pokemon
```python
# Import combinations from itertools
from itertools import combinations

# Create a combination object with pairs of Pok�mon
combos_obj = combinations(pokemon, 2)
print(type(combos_obj), '\n')

# Convert combos_obj to a list by unpacking
combos_2 = [c for c in combos_obj]
print(combos_2, '\n')

# Collect all possible combinations of 4 Pok�mon directly into a list
combos_4 = [c for c in combinations(pokemon, 4)]
print(combos_4)
```

## Comparing Pokedexes
```python
# Convert both lists to sets
ash_set = set(ash_pokedex)
misty_set = set(misty_pokedex)

# Find the Pok�mon that exist in both sets
both = ash_set.intersection(misty_set)
print(both)

# Find the Pok�mon that Ash has and Misty does not have
ash_only = ash_set.difference(misty_set)
print(ash_only)

# Find the Pok�mon that are in only one set (not both)
```

```
unique_to_set = ash_set.symmetric_difference(misty_set)
print(unique_to_set)
```

## Searching for Pokemon
```python
# Convert Brock's Pok�dex to a set
brock_pokedex_set = set(brock_pokedex)
print(brock_pokedex_set)

# Check if Psyduck is in Ash's list and Brock's set
print('Psyduck' in ash_pokedex)
print('Psyduck' in brock_pokedex_set)

# Check if Machop is in Ash's list and Brock's set
print("Machop" in ash_pokedex)
print("Machop" in brock_pokedex_set)
```

## Gathering unique Pokemon
```python
# Use find_unique_items() to collect unique Pok�mon names
uniq_names_func = find_unique_items(names)
print(len(uniq_names_func))

# Convert the names list to a set to collect unique Pok�mon names
uniq_names_set = set(names)
print(len(uniq_names_set))

# Check that both unique collections are equivalent
print(sorted(uniq_names_func) == sorted(uniq_names_set))

# Use the best approach to collect unique primary types and generations
uniq_types = set(primary_types)
uniq_gens = set(generations)
print(uniq_types, uniq_gens, sep='\n')
```

## Gathering Pokemon without a loop
```python
# Collect Pok�mon that belong to generation 1 or generation 2
gen1_gen2_pokemon = [name for name,gen in zip(poke_names, poke_gens) if gen in [1, 2]]

# Create a map object that stores the name lengths
name_lengths_map = map(len, gen1_gen2_pokemon)

# Combine gen1_gen2_pokemon and name_lengths_map into a list
gen1_gen2_name_lengths = [*zip(gen1_gen2_pokemon, name_lengths_map)]

print(gen1_gen2_name_lengths_loop[:5])
print(gen1_gen2_name_lengths[:5])
```

## Pokemon totals and averages without a loop
```python
# Create a total stats array
total_stats_np = stats.sum(axis=1)

# Create an average stats array
avg_stats_np = stats.mean(axis=1)

# Combine names, total_stats_np, and avg_stats_np into a list
poke_list_np = [*zip(names, total_stats_np, avg_stats_np)]

print(poke_list_np == poke_list, '\n')
```

```
print(poke_list_np[:3])
print(poke_list[:3], '\n')
top_3 = sorted(poke_list_np, key=lambda x: x[1], reverse=True)[:3]
print('3 strongest Pok�mon:\n{}'.format(top_3))
```

## One-time calculation loop
```python
# Import Counter
from collections import Counter

# Collect the count of each generation
gen_counts = Counter(generations)

# Improve for loop by moving one calculation above the loop
total_count = len(generations)

for gen,count in gen_counts.items():
    gen_percent = round(count / total_count * 100, 2)
    print('generation {}: count = {:3} percentage = {}'
          .format(gen, count, gen_percent))
```

## Holistic conversion loop
```python
# Collect all possible pairs using combinations()
possible_pairs = [*combinations(pokemon_types, 2)]

# Create an empty list called enumerated_tuples
enumerated_tuples = []

# Add a line to append each enumerated_pair_tuple to the empty list above
for i,pair in enumerate(possible_pairs, 1):
    enumerated_pair_tuple = (i,) + pair
    enumerated_tuples.append(enumerated_pair_tuple)

# Convert all tuples in enumerated_tuples to a list
enumerated_pairs = [*map(list, enumerated_tuples)]
print(enumerated_pairs)
```

## Bringing it all together: Pokemon z-scores
```python
# Calculate the total HP avg and total HP standard deviation
hp_avg = hps.mean()
hp_std = hps.std()

# Use NumPy to eliminate the previous for loop
z_scores = (hps - hp_avg)/hp_std

# Combine names, hps, and z_scores
poke_zscores2 = [*zip(names, hps, z_scores)]
print(*poke_zscores2[:3], sep='\n')

# Use list comprehension with the same logic as the highest_hp_pokemon code block
highest_hp_pokemon = [(name, hp, zscore) for name,hp,zscore in poke_zscores2 if zscore > 2]
print(*highest_hp_pokemon, sep='\n')
```

# 4. Basic pandas optimizations
## Iterating with .iterrows()
```python
# Print the row and type of each row
for row_tuple in pit_df.iterrows():
```

```
    print(row_tuple)
    print(type(row_tuplea))
```

## Run differentials with .iterrows()
```python
# Create an empty list to store run differentials
run_diffs = []

# Write a for loop and collect runs allowed and runs scored for each row
for i,row in giants_df.iterrows():
    runs_scored = row['RS']
    runs_allowed = row['RA']

    # Use the provided function to calculate run_diff for each row
    run_diff = calc_run_diff(runs_scored, runs_allowed)

    # Append each run differential to the output list
    run_diffs.append(run_diff)

giants_df['RD'] = run_diffs
print(giants_df)
```

## Another iterator method: .itertuples()
```python
# Loop over the DataFrame and print each row's Index, Year and Wins (W)
for row in rangers_df.itertuples():
  i = row.Index
  year = row.Year
  wins = row.W

  # Check if rangers made Playoffs (1 means yes; 0 means no)
  if row.Playoffs == 1:
    print(i, year, wins)
```

## Run differentials with .itertuples()
```python
run_diffs = []

# Loop over the DataFrame and calculate each row's run differential
for row in yankees_df.itertuples():

    runs_scored = row.RS
    runs_allowed = row.RA

    run_diff = calc_run_diff(runs_scored, runs_allowed)

    run_diffs.append(run_diff)

# Append new column
yankees_df["RD"] = run_diffs
print(yankees_df)
```

## Analyzing baseball stats with .apply()
```python
# Convert numeric playoffs to text
textual_playoffs = rays_df.apply(lambda row: text_playoffs(row["Playoffs"]), axis=1)
print(textual_playoffs)
```

## Settle a debate with .apply()
```python
```

```
# Display the first five rows of the DataFrame
print(dbacks_df.head())

# Create a win percentage Series
win_percs = dbacks_df.apply(lambda row: calc_win_perc(row['W'], row['G']), axis=1)
print(win_percs, '\n')

# Append a new column to dbacks_df
dbacks_df["WP"] = win_percs
print(dbacks_df, '\n')

# Display dbacks_df where WP is greater than 0.50
print(dbacks_df[dbacks_df['WP'] >= 0.50])
```

## Replacing .iloc with underlying arrays

```python
# Use the W array and G array to calculate win percentages
win_percs_np = calc_win_perc(baseball_df['W'].values, baseball_df['G'].values)

# Append a new column to baseball_df that stores all win percentages
baseball_df["WP"] = win_percs_np

print(baseball_df.head())
```

## Bringing it all together: Predict win percentage

```python
win_perc_preds_loop = []

# Use a loop and .itertuples() to collect each row's predicted win percentage
for row in baseball_df.itertuples():
    runs_scored = row.RS
    runs_allowed = row.RA
    win_perc_pred = predict_win_perc(runs_scored, runs_allowed)
    win_perc_preds_loop.append(win_perc_pred)

# Apply predict_win_perc to each row of the DataFrame
win_perc_preds_apply = baseball_df.apply(lambda row: predict_win_perc(row['RS'], row['RA']), axis=1)

# Calculate the win percentage predictions using NumPy arrays
win_perc_preds_np = predict_win_perc(baseball_df["RS"].values, baseball_df["RA"].values)
baseball_df['WP_preds'] = win_perc_preds_np
print(baseball_df.head())
```