Writing Efficient Python Code
by Logan Thomas and datacamp

what is efficient?
minimal completion time
minimal resource consumption

range(start, stop)
example
nums = range(0,11)
nums_list = list(nums)
print(nums_list)
[0, 1, …., 10]
*can also just add a ‘stop’ value if we want the function to assume we start at 0
ie
nums = range(11)
can also add a third argument ‘step’ for the function to follow an incremental pattern
nums = range(2,11,2)
[2,4,6,8,10]

enumerate()
creates an indexed list of objects
creates an index item pair for each item in the object provided
example
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’)]
with keyword argument ‘start’, we can tell enumerate where to start the index

map()
applies a function to each element in an object
takes two arguments
first the function
second the object you would lie to apply that function on
example
nums = [1.5, 2.3, 3.4, 4.6, 5.0]
rnd_nums = map(round, nums)
print(list(rnd_nums))

another nice example using lambda functions

```
nums = [1,2,3,4,5]
sqrd_nums = map(lambda x: x**2, nums)
print(list(sqrd_nums))
```

can use list()
or can unpack a range object like this [*range()]

Example

```
# 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, 1)]
print(indexed_names_unpack)

# Use map to apply str.upper to each element in names
names_map  = map(str.upper, names)

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

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

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

NumPy array overview
nums_np = np.array(range(5))
must contain elements of the same type
see type with .dtype
**homogeneity allows NumPy arrays to be more memory efficient and faster than Python lists

NumPy array broadcasting
another NumPy power
lists don’t support quick calculations without a function or list comprehension
NumPy’s broadcasting does allow us to do this to arrays
NumPy arrays vectorize operations, so they are performed on all elements of an object at once

NumPy array indexing
basic indexing identical to lists for one-dimensional arrays
it starts to change for higher dimensions

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

- Basic 2-D indexing (lists)

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

```
2
```

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

```
[1, 4]
```

```
# 2-D array

nums2_np = np.array(nums2)
```

- Basic 2-D indexing (arrays)

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

```
2
```

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

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

NumPy arrays also have a special technique called boolean indexing
with an array we can create a boolean mask
example
nums = [-2, -1, 0, 1, 2]
nums_np = np.array(nums)
nums_np > 0
output > array([False, False, False, True, True])
#gather only positive numbers
nums_np[nums_np > 0]
output > array([1, 2])

**No BOOLEAN indexing for lists

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

```
[1, 2]
```

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

```
[1, 2]
```

Example
# 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')

Magic commands
https://ipython.readthedocs.io/en/stable/interactive/magics.html
prefixed by the % character

here we will be using %timeit to analyze runtime
example
%timeit rand_nums = np.random.rand(1000)
output>

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

can set number of iterations you'd like to use to estimate the runtime with run argument noted with the -r flag
can set how many times you'd like the code to be executed per run with loop argument noted with the -n flag
example
%timeit -r2 -n10 rand_nums = np.random.rand(1000)
would execute our random number selection 20 times in order to estimate runtime

can use in single or multiple lines of code
single example
%timeit nums = [x for x in range(10)]
muliple example (use two %%)

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

Can save the output to a variable by using the -o flag
example
times = %timeit -o rand_nums = np.random.rand(1000)
this allows us to use attributes to see the time for each run, best time, and worst time
times.timings
times.best
times.worst

Nice sidebar - data structures
Python allows us to create with formal name or literal syntax
examples
formal_list = list()
or
literal_list = [ ]
*literal syntax has a faster runtime than formal syntax

Code profiling

a technique used to describe how long, and how often, various parts of a program are executed
*a profiler’s beauty is its ability to gather summary statistics on individual pieces of our code without using magic commands
use package line_profiler
pip install line_profiler

example
%load_ext line_profiler
%lprun -f
this will gather runtimes for individual lines of code within our function
the -f flag indicates we’d like to profile a function
*key > name of the function is passed without any parentheses
here we are specifying the name of the function we’d like to profile
%lprun -f convert_units
then provide the exact function call we’d like to profile by including any arguments that are needed
%lprun -f convert_ units convert_units(heroes, hts, wts)
output>

```
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
```

line # is the line of code
hits is how many times that line was executed
time shows time (unit is listed at top)
per hit represents the average amount of time spent executing a single line (divides time column by hits column)

Code profiling for memory usage
quick and dirty approach
example
import sys
nums_list = [*range(1000)]
sys.getsizeof(nums_list)

output > size of the object in bytes
the problem is this only works for the size of a single object
for line by line we can use the code profiler again