Apply Functions to Elements in a List

any: Check if Any Element of an Iterable is True

If you want to check if any element of an iterable is `True`, use `any`. In the code below, I use `any` to find if any element in the text is in uppercase.

```
text = "abcdE"
any(c for c in text if c.isupper())
```

```
True
```

all: Check if All Elements of an Interable Are Strings

If you want to check if all elements of an iterable are strings, use `all` and `isinstance`.

```
l = ['a', 'b', 1, 2]

all(isinstance(item, str) for item in l)
```

```
False
```

filter: Get the Elements of an Iterable that a Function Returns True

If you want to get the elements of an iterable that a function returns true, use `filter`.

In the code below, I use the `filter` method to get items that are fruits.

```
def get_fruit(val: str):
    fruits = ['apple', 'orange', 'grape']
    if val in fruits:
        return True
    else:
        return False

items = ['chair', 'apple', 'water', 'table', 'orange']
fruits = filter(get_fruit, items)
print(list(fruits))
```

```
['apple', 'orange']
```

map method: Apply a Function to Each Item of an Iterable

If you want to apply the given function to each item of a given iterable, use `map`.

```
nums = [1, 2, 3]
list(map(str, nums))
```

```
['1', '2', '3']
```

```
def multiply_by_two(num: float):
    return num * 2

list(map(multiply_by_two, nums))
```

```
[2, 4, 6]
```

Get Elements

random.choice: Get a Randomly Selected Element from a Python List

Besides getting a random number, you can also get a random element from a Python list using random. In the code below, “stay at home” was picked randomly from a list of options.

```
import random

to_do_tonight = ['stay at home', 'attend party', 'do
exercise']

random.choice(to_do_tonight)
```

```
'attend party'
```

random.sample: Get Multiple Random Elements from a Python List

If you want to get n random elements from a list, use `random.sample`.

```
import random

random.seed(1)
nums = [1, 2, 3, 4, 5]
random_nums = random.sample(nums, 2)
random_nums
```

```
[2, 1]
```

heapq: Find n Max Values of a Python List

If you want to extract n max values from a large Python list, using `heapq` will speed up the code.

In the code below, using heapq is more than 2 times faster than using sorting and indexing. Both methods try to find the max values of a list of 10000 items.

```
import heapq
import random
from timeit import timeit

random.seed(0)
l = random.sample(range(0, 10000), 10000)

def get_n_max_sorting(l: list, n: int):
    l = sorted(l, reverse=True)
    return l[:n]

def get_n_max_heapq(l: list, n: int):
    return heapq.nlargest(n, l)
```

```
expSize = 1000
n = 100
time_sorting = timeit("get_n_max_sorting(l, n)",
number=expSize,
                      globals=globals())
time_heapq = timeit('get_n_max_heapq(l, n)', number=expSize,
                    globals=globals())

ratio = round(time_sorting/time_heapq, 3)
print(f'Run {expSize} experiments. Using heapq is {ratio}
times'
' faster than using sorting')
```

```
Run 1000 experiments. Using heapq is 2.827 times faster than
using sorting
```

Good Practices

Stop using = operator to create a copy of a Python list. Use copy method instead

When you create a copy of a Python list using the = operator, a change in the new list will lead to the change in the old list. It is because both lists point to the same object.

```
>>> l1 = [1, 2, 3]
>>> l2 = l1
>>> l2.append(4)
```

```
>>> l2
```

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

```
>>> l1
```

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

Instead of using = operator, use `copy()` method. Now your old list will not change when you change your new list.

```
>>> l1 = [1, 2, 3]
>>> l2 = l1.copy()
>>> l2.append(4)
```

```
>>> l2
```

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

```
>>> l1
```

```
[1, 2, 3]
```

Enumerate: Get Counter and Value While Looping

Are you using `for i in range(len(array))` to access both the index and the value of the array? If so, use `enumerate` instead. It produces the same result but it is much cleaner.

```
arr = ['a', 'b', 'c', 'd', 'e']

# Instead of this
for i in range(len(arr)):
    print(i, arr[i])
```

```
0 a
1 b
2 c
3 d
4 e
```

```
# Use this
for i, val in enumerate(arr):
    print(i, val)
```

```
0 a
1 b
2 c
3 d
4 e
```

Difference between list append and list extend

If you want to add a list to another list, use the `append` method. To add elements of a list to another list, use the `extend` method.

```
# Add a list to a list
>>> a = [1, 2, 3, 4]
>>> a.append([5, 6])
>>> a
```

```
[1, 2, 3, 4, [5, 6]]
```

```
>>> a = [1, 2, 3, 4]
>>> a.extend([5, 6])

>>> a
```

```
[1, 2, 3, 4, 5, 6]
```

Interaction Between 2 Lists

set.intersection: Find the Intersection Between 2 Sets

If you want to get the common elements between 2 lists, convert lists to sets then use `set.intersection` to find the intersection between 2 sets.

```
requirement1 = ['pandas', 'numpy', 'statsmodel']
requirement2 = ['numpy', 'statsmodel', 'sympy', 'matplotlib']

intersection = set.intersection(set(requirement1),
set(requirement2))
list(intersection)
```

```
['statsmodel', 'numpy']
```

Set Difference: Find the Difference Between 2 Sets

If you want to find the difference between 2 lists, turn those lists into sets then apply the `difference()` method to the sets.

```
a = [1, 2, 3, 4]
b = [1, 3, 4, 5, 6]
```

```
# Find elements in a but not in b
diff = set(a).difference(set(b))
print(list(diff))
```

```
[2]
```

```
# Find elements in b but not in a
diff = set(b).difference(set(a))
print(list(diff))  # [5, 6]
```

```
[5, 6]
```

Join Iterables

join method: Turn an Iterable into a Python String

If you want to turn an iterable into a string, use `join()`.

In the code below, I join elements in the list fruits using “, “.

```
fruits = ['apples', 'oranges', 'grapes']

fruits_str = ', '.join(fruits)

print(f"Today, I need to get some {fruits_str} in the grocery store")
```

```
Today, I need to get some apples, oranges, grapes in the grocery store
```

Zip: Associate Elements from Two Iterators based on the Order

If you want to associate elements from two iterators based on the order, combine `list` and `zip`.

```
nums = [1, 2, 3, 4]
string = "abcd"
combinations = list(zip(nums, string))
combinations
```

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

Zip Function: Create Pairs of Elements from Two Lists in Python

If you want to create pairs of elements from two lists, use `zip`. `zip()` function takes iterables and aggregates them in a tuple.

You can also unzip the list of tuples by using `zip(*list_of_tuples)`.

```
nums = [1, 2, 3, 4]
chars = ['a', 'b', 'c', 'd']

comb = list(zip(nums, chars))
comb
```

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

```
nums_2, chars_2 = zip(*comb)
nums_2, chars_2
```

```
((1, 2, 3, 4), ('a', 'b', 'c', 'd'))
```

Unpack Iterables

How to Unpack Iterables in Python

To assign items of a Python iterables (such as list, tuple, string) to different variables, you can unpack the iterable like below.

```
nested_arr = [[1, 2, 3], ["a", "b"], 4]
num_arr, char_arr, num = nested_arr
```

```
num_arr
```

```
[1, 2, 3]
```

```
char_arr
```

```
['a', 'b']
```

Extended Iterable Unpacking: Ignore Multiple Values when Unpacking a Python Iterable

If you want to ignore multiple values when unpacking a Python iterable, add * to _ as shown below.

This is called “Extended Iterable Unpacking” and is available in Python 3.x.

```
a, *_, b = [1, 2, 3, 4]
print(a)
```

```
1
```

```
b
```

```
4
```

```
_
```

```
[2, 3]
```