* Copy object:

Copy module, copy.copy //shallow, copy.deepcopy() //deep.

Dict.copy() //shallow.

For list-like objects, slicing can be used to create a copy:

* original = {'a': 1, 'b': 2} copies = [original] \* 4 # Create a list of 4 references

all references the same copy of original. Changing original[‘a’] = 5 will change everything.

copies[0]['a'] = 99

print(copies) # [{'a': 99, 'b': 2}, {'a': 99, 'b': 2}, {'a': 99, 'b': 2}, {'a': 99, 'b': 2}]

 When you assign an object to a variable, you are assigning a reference to the object, not a copy of it.

 The \* operator replicates these references, not the actual object.

If you need independent objects, use a method that explicitly creates separate copies for each element.

copies = [copy.deepcopy(original) for \_ in range(4)]

* Memoization

Memoization is useful in situations where previously calculated results can be reused. It is particularly effective in recursive problems, especially those involving overlapping subproblems, where the same calculations are repeated multiple times.

* Merge two lists, or a list and a value, will result in a flat list,

list1 = [1, 2, 3] list2 = [4, 5, 6] merged\_list = list1 + list2

* list1 = [[1, 2], [3, 4]] list2 = [[5, 6], [7, 8]]

merged\_list = list1 + list2

[[1, 2], [3, 4], [5, 6], [7, 8]]

**Note: unpacking lists will result in one list:**

list1 = [1, 2, 3] list2 = [4, 5, 6]

merged\_list = [\*list1, \*list2]

print(merged\_list) # [1, 2, 3, 4, 5, 6]

* can remove a node from list based on its value.

parent\_option\_list = [1, 2, 3, 4]

choice = 2

parent\_option\_list.remove(choice) # Removes the first occurrence of 2

* **slicing** a list in Python **always creates a new copy** of the portion of the list you slice
* to initialize an empty set, use myset = set()

or my\_set = {1, 2, 3, 4}

* Decorators
* List comprehension
* Dictionary comprehension
* Magic functions
* Collections
* Pickle
* Count

 Sets **do not** have a count() method since they only store unique elements.

 Use list.count() if you want to count occurrences of an element in a list.

 For general counting across any iterable, use collections.Counter.

from collections import Counter

my\_list = [1, 2, 2, 3, 4, 1, 2]

counter = Counter(my\_list)

print(counter[2]) # Output: 3 (counts occurrences of 2 in the list)

* string.find(substring) returns the lowest index, if not find, returns -1
* strings are immutable, to replace a piece, can do

new\_string = s[:start\_index] + replacement + s[end\_index:]

* print("inner x:", x, "y:", y)

print with comma introduce space automatically

* python advanced topics: good sites:  
  [Python Closures (With Examples)](https://www.programiz.com/python-programming/closure)
* Nested function in Python
* Closure

 closure is a nested function that helps us access the outer function's variables even after the outer function is closed

def greet():

name = "John" # variable defined outside the inner function

return lambda: "Hi " + name # return a nested anonymous function

message = greet() # call the outer function

print(message()) # call the inner function

* nonlocal: scope the variable to the same variable in the parent level

global: scope the variable to the same variable in the global level. Note: if there is parent level variable, the parent level variable is skipped

* parameters passed in from a function, it is saved in the closure of the inner function.

def make\_multiplier\_of(n):

def multiplier(x):

def inner\_multiplier(y):

print("x:", x, "y:",y, "n:", n)

return x \* y \* n

return inner\_multiplier # Ensure inner\_multiplier is returned

return multiplier # Ensure multiplier is returned

times3 = make\_multiplier\_of(3)

print(times5(3)(1))

output:

x: 3 y: 1 n: 5

15

* All function objects have a \_\_closure\_\_ attribute that returns a [tuple](https://www.programiz.com/python-programming/tuple) of cell objects if it is a closure function.
* callable