# wiegman_lab01

August 21, 2024

[ ]:

```
# Problem 1: Understanding Integers and Floats
## Part A
def add_and_multiply(a,b):
    return (a+b, a*b)

## Part B
def convert_types(a):
    if type(a) is float:
        return int(a)
    elif type(a) is int:
        return float(a)
    else:
        return "Not float or int!"
```

[ ]:

```
# Problem 2: Working with Lists
## Part A
intlist = [1,2,3,4,5,6,7,8,9,10]

def manipulate_list(input):
    input.append(11)
    input.remove(input[2])
    input.reverse()
    return input

print(manipulate_list(intlist))

## Part B
def list_length(input):
    return len(input)

print(list_length(intlist))
```

```
[11, 10, 9, 8, 7, 6, 5, 4, 2, 1]
10
```

[ ]:

```
# Problem 3: Working with Tuples
## Part A
fruit_tuple = ("apple", "blueberry", "cherry", "date", "elderberry")
```

```
def access_fruits(input):
    second = input[1]
    fourth = input[3]
    input[2] = "blackberry" # this does not work, because tuples cannot be␣
 ↪changed after instantiation

access_fruits(fruit_tuple)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 10
      7     fourth = input[3]
      8     input[2] = "blackberry" # this does not work, because tuples cannot␣
 ↪be changed after instantiation
---> 10 access_fruits(fruit_tuple)

Cell In[3], line 8, in access_fruits(input)
      6 second = input[1]
      7 fourth = input[3]
----> 8 input[2] = "blackberry"

TypeError: 'tuple' object does not support item assignment
```

```
[ ]: # Problem 4: Understanding Dictionaries
     ## Part A
     student_scores = {
         "alice" : 100,
         "bob"   : 90,
         "cathy" : 80
     }
     def dict_operations(input):
         input["dale"] = 70
         input["alice"] = 110
         input.pop("cathy")
         print(f"Bob's score is: {input["bob"]}")

     dict_operations(student_scores)
     print(student_scores)

     # Part B
     print(f'Using .get() gives {student_scores.get("bad_key")}')
```

```
Bob's score is: 90
{'alice': 110, 'bob': 90, 'dale': 70}
Using .get() gives None
```

[ ]:

```
print(f'Using [] gives {student_scores["bad_key"]}')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[5], line 1
----> 1 print(f'Using [] gives {student_scores["bad_key"]}')

KeyError: 'bad_key'
```

[ ]:

```
# Problem 5: Working with Sets
A = {1,2,3,4}
B = {3,4,5,6}

def set_operations(set1, set2):
    union = set1.union(set2)
    inter = set1.intersection(set2)
    difff = set1.difference(set2)
    chksb = set1.issubset(set2)
```

[ ]:

```
# Problem 6: Arrays and Numpy
import numpy as np

## Part A
array1d = np.array([0,1,2,3,4,5,6,7,8,9])

def array_operations(input):
    input = 2*input
    return input + 3

## Part B
array2d = np.array([[0,1],[2,3]])

def matrix_operations(input):
    return input.transpose() + input
```

[ ]:

```
# Problem 7: Working with Strings

def string_manipulation(input):
    uppercase = input.upper()
    number_of_K = input.count("K")
    rev_string = "".join(reversed(input))
```

[ ]:

```
# Problem 8: Combining Lists and Dictionaries
## Part A
```

```
students = [
    {"name": "alice", "age": 10, "score": 100},
    {"name": "bob",   "age": 11, "score":  90},
    {"name": "cathy", "age":  9, "score":  80}
]

## Part B
def average_score(input):
    nstudents = 0
    total_points = 0

    for student in input:
        nstudents += 1
        total_points += student["score"]

    return total_points/nstudents

average_score(students)
```

```
[ ]: 90.0
```

```
[ ]: # Problem 9: Combining Data Structures for AI-Like Processing
     ## Part A
     sample_text = "Hey guys, did you know that in terms of human companionship,␣
      ↪Flareon is objectively the most huggable Pokemon? While their maximum␣
      ↪temperature is likely too much for most, they are capable of controlling it,␣
      ↪so they can set themselves to the perfect temperature for you. Along with␣
      ↪that, they have a lot of fluff, making them undeniably incredibly soft to␣
      ↪touch. But that's not all, they have a very respectable special defense stat␣
      ↪of 110, which means that they are likely very calm and resistant to␣
      ↪emotional damage. Because of this, if you have a bad day, you can vent to it␣
      ↪while hugging it, and it won't mind. It can make itself even more endearing␣
      ↪with moves like Charm and Baby Doll Eyes, ensuring that you never have a␣
      ↪prolonged bout of depression ever again."
     cleaned_text = sample_text.replace(".","").replace(",","").replace("?","").
      ↪lower()
     words = cleaned_text.split(" ")
     word_freq = {}

     for word in sorted(list(set(words))):
         word_freq[word] = words.count(word)

     print(word_freq)
     print("\n") # for neatness in output

     ## Part B
     tuple_freq = []
```

```python
for word in word_freq:
    tuple_freq.append((word, word_freq[word]))

tuple_freq = sorted(tuple_freq, key =lambda x: x[1])
print(tuple_freq)
```

```
{'110': 1, 'a': 4, 'again': 1, 'all': 1, 'along': 1, 'and': 3, 'are': 2, 'baby':
1, 'bad': 1, 'because': 1, 'bout': 1, 'but': 1, 'calm': 1, 'can': 3, 'capable':
1, 'charm': 1, 'companionship': 1, 'controlling': 1, 'damage': 1, 'day': 1,
'defense': 1, 'depression': 1, 'did': 1, 'doll': 1, 'emotional': 1, 'endearing':
1, 'ensuring': 1, 'even': 1, 'ever': 1, 'eyes': 1, 'flareon': 1, 'fluff': 1,
'for': 2, 'guys': 1, 'have': 4, 'hey': 1, 'huggable': 1, 'hugging': 1, 'human':
1, 'if': 1, 'in': 1, 'incredibly': 1, 'is': 2, 'it': 5, 'itself': 1, 'know': 1,
'like': 1, 'likely': 2, 'lot': 1, 'make': 1, 'making': 1, 'maximum': 1, 'means':
1, 'mind': 1, 'more': 1, 'most': 2, 'moves': 1, 'much': 1, 'never': 1, 'not': 1,
'objectively': 1, 'of': 6, 'perfect': 1, 'pokemon': 1, 'prolonged': 1,
'resistant': 1, 'respectable': 1, 'set': 1, 'so': 1, 'soft': 1, 'special': 1,
'stat': 1, 'temperature': 2, 'terms': 1, 'that': 4, "that's": 1, 'the': 2,
'their': 1, 'them': 1, 'themselves': 1, 'they': 5, 'this': 1, 'to': 4, 'too': 1,
'touch': 1, 'undeniably': 1, 'vent': 1, 'very': 2, 'which': 1, 'while': 2,
'with': 2, "won't": 1, 'you': 5}


[('110', 1), ('again', 1), ('all', 1), ('along', 1), ('baby', 1), ('bad', 1),
('because', 1), ('bout', 1), ('but', 1), ('calm', 1), ('capable', 1), ('charm',
1), ('companionship', 1), ('controlling', 1), ('damage', 1), ('day', 1),
('defense', 1), ('depression', 1), ('did', 1), ('doll', 1), ('emotional', 1),
('endearing', 1), ('ensuring', 1), ('even', 1), ('ever', 1), ('eyes', 1),
('flareon', 1), ('fluff', 1), ('guys', 1), ('hey', 1), ('huggable', 1),
('hugging', 1), ('human', 1), ('if', 1), ('in', 1), ('incredibly', 1),
('itself', 1), ('know', 1), ('like', 1), ('lot', 1), ('make', 1), ('making', 1),
('maximum', 1), ('means', 1), ('mind', 1), ('more', 1), ('moves', 1), ('much',
1), ('never', 1), ('not', 1), ('objectively', 1), ('perfect', 1), ('pokemon',
1), ('prolonged', 1), ('resistant', 1), ('respectable', 1), ('set', 1), ('so',
1), ('soft', 1), ('special', 1), ('stat', 1), ('terms', 1), ("that's", 1),
('their', 1), ('them', 1), ('themselves', 1), ('this', 1), ('too', 1), ('touch',
1), ('undeniably', 1), ('vent', 1), ('which', 1), ("won't", 1), ('are', 2),
('for', 2), ('is', 2), ('likely', 2), ('most', 2), ('temperature', 2), ('the',
2), ('very', 2), ('while', 2), ('with', 2), ('and', 3), ('can', 3), ('a', 4),
('have', 4), ('that', 4), ('to', 4), ('it', 5), ('they', 5), ('you', 5), ('of',
6)]
```

[ ]:

```python
# Problem 10: Simulating Basic AI Concepts Using Python Data Structures
## Part A
def perceptron(weights: list, inputs: list, bias: float):
    output = 0
    for i in range(len(weights)):
        output += (weights[i] * inputs[i])
    return output + bias

## Part B
def update_weights(orig_weights: list, inputs: list, orig_bias: float,␣
 ↪expected_output: float):
    orig_output = perceptron(orig_weights, inputs, orig_bias)
    error = expected_output - orig_output

    # Very simple & naive update algorithm: add
    #   half error to bias, half error across weights
    new_bias = orig_bias + error/2
    n_terms = len(inputs)
    new_weights = list(map(lambda x: x*((error/2)/n_terms), orig_weights))
    return (new_bias, new_weights)
```