CS35L: Week 2 Python Worksheet Solutions

Python

When tackling all these problems, make sure to write code well: follow the snake_case convention for variables and functions instead of camelCase. Adhere to PEP 8 guidelines. Write comments as necessary, and make sure your variable and function names are descriptive. While correctness is the highest priority, if you can, try to be pythonic and keep your solutions as short and simple as possible.

Tips

- **Style.** The Python programming language has its own preferred "coding style" – kind of like its own personality – which you can find in PEP 8. Some notable examples of the PEP 8 guidelines:
  - Use `snake_case` for variables and functions instead of `camelCase`.
  - Function and variable names should be descriptive. For example, the function name `get_server_status()` is better than `status()`.
  - … and many more. See the full document for details!
- **For-loops.** For-loops in Python *iterate over lists* (unlike in C++). If you haven't seen this idea before, spend some time playing around with it! You'll soon realize that Python for-loops are a lot more convenient than C++ for loops.

Basic Problems

1. Write a function called `get_max(num0, num1)` that takes in two numbers and returns the greater of the two. Don't worry about type-checking; assume that num0 and num1 are both `int`s or `float`s.
   For example, `get_max(8, 2)` should return 8.

   Answer:

   ```
   def get_max(x, y):
     return x if x > y else y
   ```

2. Write a function called `sum_range(lower, upper)` that takes in a lower bound and upper bound and returns the sum of [lower bound, upper bound). If the lower bound is greater than or equal to the upper bound, return 0. Assume both arguments are always integers.
   For example, `sum_range(3, 6)` should return 12.

   Answer:

   ```
   def sum_range(lower, upper):
     result = 0
     for i in range(lower, upper):
   ```

```
    result += i
  return result
```

3. Mergesort is a divide-and-conquer algorithm. In the conquering portion, you need to merge two smaller sorted lists into one large sorted list. Write a function named `merge_lists(list1, list2)` which takes in two **sorted** lists and merges them into one larger, sorted list.

   The lists may have different lengths. Don't worry about type-checking.
   For example, `merge_lists([1, 3, 5, 6], [2, 7])` should return `[1, 2, 3, 5, 6, 7]`.

   Then, write a function `merge_sort(unsorted)` to merge sort an unsorted list.
   Answer:

```
#!/bin/python3


def merge_lists(list1, list2):
      result = []

      while len(list1) and len(list2):
      if list1[0] < list2[0]:
            result.append(list1[0])
            list1 = list1[1:]
      else:
            result.append(list2[0])
            list2 = list2[1:]
      result += list1
      result += list2

      return result


def merge_sort(unsorted):
      if len(unsorted) <= 1:
      return unsorted

      l1 = unsorted[:len(unsorted)//2]
      l2 = unsorted[len(unsorted)//2:]
```

```
    l1 = merge_sort(l1)
    l2 = merge_sort(l2)

    return merge_lists(l1, l2)


if __name__ == '__main__':
    numbers = [1, 2, 4, 1, 3, 6, 10, 15, 21, 8, 16, 32]
    print(merge_sort(numbers))
```

Other Python Problems:

1. Write a class named `RegularPolygon`. Each instance should have `number_of_sides` and `side_length` data members, which are passed in at initialization time. It should also have a member function `get_perimeter()` that calculates the shape's perimeter.
   For example, the following code snippet:

```
pentagon = RegularPolygon(5, 2.7)
print(f"This pentagon has {pentagon.number_of_sides} sides, a
side length of {pentagon.side_length}, and a perimeter of
{pentagon.get_perimeter()}.")
```

   should output: "This pentagon has 5 sides, a side length of 2.7, and a perimeter of 13.5."

   Answer:

```
class RegularPolygon:
  def __init__(self, number_of_sides, side_length):
    self.number_of_sides = number_of_sides
    self.side_length = side_length
  def get_perimeter(self):
    return (self.number_of_sides * self.side_length)
```

2. Write a function called `is_two_summable(nums, target)` that takes a list of numbers and a target sum. It returns `True` if there exist two numbers in the list that sum to the target, and `False` otherwise.
   a. Example: `is_two_summable([1, 2, 3, 4], 7)` should return `True` because 3 + 4 is 7.

b. Example: `is_two_summable([1, 2, 3, 4], 10)` should return `False`, because there are no two numbers in the list that sum to 10.
c. Bonus: What is the time complexity of your algorithm? If *N* is the length of `nums`, then the optimal solution runs in *O(N)* time.
d. Hint: If you're trying to optimize your solution, it might be helpful to look into Python sets.

Solution:

```
def is_two_summable(numbers, target):
    num_set = set()
    for num in numbers:
        if num in num_set:
            return True
        num_set.add(target - num)
    return False
```

Intermediate problems:

1. In Python 3.10, we have the introduction of structural pattern matching. In most coding languages, an alternative to if-else statements are switch statements. In Python, this has just been implemented via structural matching. Let us create a text-based game where the user can choose to move around a map using `move` direction commands, and do personal actions using `do` commands. Write a structural pattern match to mimic this behavior. Hint: You can run Python 3.10 on SEASnet using `python3.10`.

```
def play_game():
        play = True
        directions = ["North", "South", "East", "West"]
        while play:
        command = input()
        match command.split():
        case ["quit", *args] | ["end", *args]:
                play = False
        case ["go", direction] if direction in directions:
                print("You went", direction)
        case ["do", *something]:
                if len(something) == 0:
                print("You did nothing.")
                elif len(something) == 1:
                print("You did ", something[0], ".", sep="\n")
                elif len(something) == 2:
                print("You did ", something[0], " and ", something[1], ".", sep="")
```

```
                else:
                print("You did", end=" ")
                for _ in range(len(something) - 1):
                print(something[_], end=", ")
                print("and ", something[len(something) - 1], ".", sep="")
        case _:
                print("I'm not sure I understand...")
                print("Options:", "do [*something]", "go [direction]", "quit", "end",
sep="\n")



if __name__ == '__main__':
        play_game()
```

2. One of the most useful modules in Python outside of numpy is the argparse module, which allows command line parsing for optional parameters. Create a calculator that takes in two integers and outputs their sum, but can do other arithmetic operations if optional parameters are given.

```
#!/bin/python3

import argparse


def parse():
        parser = argparse.ArgumentParser(prog="Calculator script")
        parser.add_argument('numbers', nargs=2, type=int)
        parser.add_argument('--sum', '-s', action='store_true', required=False,
default=True)
        parser.add_argument('--product', '-p', action='store_true', required=False,
default=False, dest='times')

        args = parser.parse_args()
        print(args)
        a = args.numbers[0]
        b = args.numbers[1]
        if args.times:
                c = a * b
                print("Product of", "and", b, "is", c)
        elif args.sum:
                c = a + b
```

```
                print("Sum of", "and", b, "is", c)
        return(c)


if __name__ == '__main__':
        parse()
```