CS521 – Homework 1 – Alexander Bean

1. The PEMDAS Rule

In the following problems (1a-1e), I followed the PEMDAS rule to explain the calculations done. The steps start with Parentheses, then Exponents, Multiplication/Division, and Addition/Subtraction. I broke it down by (mostly) individual steps to clearly show the process, so those steps should provide the explanation necessary.

a. 8+2×(3**2 –4)/2

In [10]:
```
# a. 8+2×(3**2 -4)/2

# 8 + 2 * (3 ** 2 - 4) / 2
# 8 + 2 * (9 - 4) / 2
# 8 + 2 * 5 / 2
# 8 + 10 / 2
# 8 + 5
# 13

8 + 2 * (3 ** 2 - 4) / 2
```

Out[10]: 13.0

b. ((6+4)/2)×5–3**2 +(8–5×2)

In [11]:
```
# b. ((6+4)/2)×5-3**2 +(8-5×2)

# ((6 + 4) * 5 - 3 ** 2) + (8 - 5 * 2)
# (10 * 5 - 3 ** 2) + (8 - 5 * 2)
# (10 * 5 - 9) + (8 - 5 * 2)
# (50 - 9) + (8 - 10)
# 41 + -2
# 39

((6 + 4) * 5 - 3 ** 2) + (8 - 5 * 2)
```

Out[11]: 39

c. (12/(22 +2))×(3+5)−7**

In [5]:
```
# c. (12/(2**2 +2))×(3+5)−7

# (12 / (2 ** 2 + 2)) * (3 * 5) - 7
# (12 / (4 + 2)) * (3 * 5) - 7
# (12 / 6) * (3 * 5) - 7
# 2 * 15 - 7
# 30 - 7
# 23

(12 / (2 ** 2 + 2)) * (3 * 5) - 7
```

Out[5]: 23.0

d. 15−3×(4+6)/32+8**

In [8]:
```
# d. 15−3×(4+6)/3**2+8

# 15 - 3 * (4 + 6) / 3 ** 2 + 8
# 15 - 3 * 10 / 3 ** 2 + 8
# 15 - 3 * 10 / 9 + 8
# 15 - 30 / 9 + 8
# 15 - 3.33 + 8
# 11.66 + 8
# 19.66
#
# note that the decimals in this problem are repeating

15 - 3 * (4 + 6) / 3 ** 2 + 8
```

Out[8]: 19.666666666666664

e. (5×(6–4)2+2)/(2+3**2)–10/2**2+2**

In [21]:

```
# e. (5×(6–4)**2+2)/(2+3**2)–10/2**2+2

# (5 * (6 - 4) ** 2 + 2) / (2 + 3 ** 2) - 10 / 2 ** 2 + 2
# (5 * 2 ** 2 + 2) / (2 + 3 ** 2) - 10 / 2 ** 2 + 2
# (5 * 4 + 2) / (2 + 9) - 10 / 2 ** 2 + 2
# (20 + 2) / (2 + 9) - 10 / 2 ** 2 + 2
# 22 / 11 - 10 / 2 ** 2 + 2
# 22 / 11 - 10 / 4 + 2
# 2 - 10 / 4 + 2
# 2 - 2.5 + 2
# -0.5 + 2
# 1.5

(5 * (6 - 4) ** 2 + 2) / (2 + 3 ** 2) - 10 / 2 ** 2 + 2
```

Out[21]: 1.5

2. Strings | Questions from Chapter 4

Question 1

In [10]:

```
# CHAPTER 4, QUESTION 1

# Given the string "Monty Python":

string = "Monty Python"

# i. Write an expression to print the first character

string_first = string[0]

print(f"\nThe FIRST letter of '{string}' is: \t{string_first}")

# ii. Write an expression to print the last character.

string_last = string[-1]
print(f"\nThe LAST letter of '{string}' is: \t{string_last}")

# iii. Write an expression including len() to print the last character.

string_last_w_len = string[len(string) - 1]
print(f"\nUsing len() to print the LAST letter:\t{string_last_w_len}")

# iv. Write an expression that prints "Monty".

string_monty = string[0:5]
print(f"\nThis expression uses slicing to print: \t{string_monty}")
```

```
The FIRST letter of 'Monty Python' is:  M

The LAST letter of 'Monty Python' is:   n

Using len() to print the LAST letter:   n

This expression uses slicing to print:  Monty
```

In this problem, I utilized indexing/slicing to print the requested text. I also used the length of the string to define the values to slice itself with, which proved useful in solving future problems.

Question 2

In [11]:

```
# CHAPTER 4, QUESTION 2

# Given the string "homebody":

string = "homebody"

# a. Write an expression using slicing to print “home”

string_home = string[:4]
print(f"\nThis expression prints 'home':\t{string_home}")

# b. Write an expression using slicing to print “body”

string_body = string[-4:]
print(f"\nThis expression prints 'body':\t{string_body}")
```

```
This expression prints 'home':  home

This expression prints 'body':  body
```

Since the specified string is 8 characters long, I sliced it based on the requested text. Using [:4] will print up to but not including index 4, so it will return 0, 1, 2, 3 which are h, o, m, e. Inversely, using [-4:] starts at 4 index values from the end of the string, which would return 4, 5, 6, 7 which are b, o, d, y. This can also be achieved by using [4:], which will print anything after (and including) the 4^{th} index.

Question 3

In [42]:

```
# CHAPTER 4, QUESTION 3

# Given a string S with even length

S = "even"

# a. Write an expression to print out the first half of the string

S_first_half = S[ : int(len(S)/2)]

print(f"\nThe first half of '{S}' is:\t{S_first_half}")

# b. Write an expression to print out the second half of the string

S_second_half = S[int(len(S)/2) : ]

print(f"\nThe second half of '{S}' is:\t{S_second_half}")
```

```
The first half of 'even' is:    ev

The second half of 'even' is:   en
```

For the first half: Using the len() function, I can identify the length of the string and use it for slicing. In this scenario, the length would be 4. The index essentially means [0:2]. This will slice the string starting from the beginning up to (but not including) the halfway point.

Additionally, I convert the value created by len() to an integer, since it defaulted to float. Decimal point values in the indexing raises an error.

For the second half: The index essentially means [2:], which will print out the string starting from the halfway point to the end.

Question 4

In [43]:

```
# CHAPTER 4, QUESTION 4

# Given a string S with odd length

S = "odd"

# a. Write an expression to print the middle character

S_middle_char = S[int(len(S)/2)]

print(f"\nThe middle char in '{S}' is:\t{S_middle_char}")

# b. Write an expression to print the string up to, but not including, the middle character (i.e. the first

S_first_half = S[ : int(len(S)/2)]

print(f"\nThe first half of '{S}' is:\t{S_first_half}")

# c. Write an expression to print the string from the middle character to the end (not including the middle

S_second_half = S[int(len(S)/2) + 1 : ]

print(f"\nThe second half of '{S}' is:\t{S_second_half}")
```

```
The middle char in 'odd' is:    d

The first half of 'odd' is:     o

The second half of 'odd' is:    d
```

I utilized the same methods as Question 3 when tackling this problem. I used len() on the string to determine index values. This value will come back as a decimal value since, when an odd number is divided in half, you end up with a decimal / float value number. Using the int() function truncates the decimals and identifies the middle index value.

Printing the string up to, but not including, the middle character is the same answer and reasoning as in Question 3.

Printing the second half of the string but not including the middle character was a similar approach. I added the + 1 to the index since int() truncates/rounds DOWN and not up, and we need the index to begin at the rounded up number.

Question 13

In [44]:
```
# CHAPTER 4, QUESTION 13

# We know that writing the following code:
#   print("I like writing in Python.")
#   print("It is so much fun.")
# will result in:
#   I like writing in Python.
#   It is so much fun.
# When executed.

# However, can you manage to do this same task with only one line of code?

print("I like writing in Python." + " \n" + "It is so much fun.")
print("I like writing in Python.", "\nIt is so much fun.")
```

```
I like writing in Python.
It is so much fun.
I like writing in Python.
It is so much fun.
```

Yes, you can. The print() statement can take in many, many parameters or statements. I included two examples of how you can achieve this through concatenation of strings and through entering multiple statements.

Question 15

In [45]:
```
# CHAPTER 4, QUESTION 15

# It is possible to combine string methods in one expression.
# Given the expression s = "CAT", what is s.upper().lower()?

s = "CAT"

s.upper().lower()
```

Out[45]:
```
'cat'
```

The two expressions will be executed from left-to-right. In this scenario, upper() is executed and turns s into “CAT, then lower() is executed and turns s into “cat”.

Question 17

In [46]:
```
# CHAPTER 4, QUESTION 17

# Two string methods find where a character is in a strong: find and index

# a. Both work the same if a character is found, but behave differently if the character is NOT found.
#    Describe the difference in how they behave when the character is not found.

# Uncommenting the lines below will raise an error
# string = "test"
# string.index("Q")

string = "test"
print(string.find("Q"))

# b. The find and index methods are not limited to finding single characters. They can search for substring
#    Given s = “Topkapi”, what does s.find(“kap”) print? Describe the rule for what find prints.

s = "Topkapi"

s.find("kap")
```

```
-1
```

Out[46]: 3

When an item is not found using index(), it will raise an error saying the substring was not found. If you uncomment those two lines, it will raise that error. However, if an item cannot be found using find(), it will return a negative index value instead.

In the Topkapi example: this prints 3, and find() will print where the first instance of the string is located. The "k" in "kap" is the 4th value in the string, which would be the 3rd index value.

Question 19

In [47]:
```
# CHAPTER 4, QUESTION 19

# Convert a string that is all capitals into a string where only the first letters are capitals.
# For example, convert "NEW YORK" to "New York".

string = "NEW YORK"

string = string.title()

string
```

Out[47]: 'New York'

To achieve this, I can use the .title() expression. The title() expression will capitalize the first letter in each work and lower the rest.

Question 27

In [48]:
```
# CHAPTER 4, QUESTION 27

# (Reversing a string) Given a string, say X = "Alan Turing"
# write an expression to reverse it to get a string, Y = "gniruT nalA"

X = "Alan Turing"

Y = X[::-1]

print(f"The string '{X}' reversed is: '{Y}'")
```

```
The string 'Alan Turing' reversed is: 'gniruT nalA'
```

By using the "Step" part of indexing/slicing, I can step through the string backwards (reversing it)

Question 35

In [49]:
```
# CHAPTER 4, QUESTION 35

# Using only the string find method, display the integer 23 from the string "Lebron James"

string = "Lebron James"

string.find("s") + string.find("s") + string.find("e")

# To verify it is an int, uncomment the line below.
# type(string.find("s") + string.find("s") + string.find("e"))
```

Out[49]: 23

The find() method prints the index value where the parameter is found. To get 23, I used the find() method on specific letters of the string and added them. The values of those specific letters, when added, produce the integer 23.

11 + 11 + 1 = 23

3. Lists and Tuples | Questions from Chapter 7

Question 6

In [50]:
```
# CHAPTER 7, QUESTION 6

list1 = [1, 2, 99]
list2 = list1
list3 = list2
list1 = list1.remove(1)

# a. What is printed?

print(list3)

# b. How can you change the code so list3 is unchanged?

list1 = [1, 2, 99]
list2 = list1
list3 = tuple(list2)
list1 = list1.remove(1)

print(list3)

list3 = list(list3)
print(list3)
```

```
[2, 99]
(1, 2, 99)
[1, 2, 99]
```

This prints the same list of [1, 2, 99] because lists are mutable (aka they can be changed/altered).

To change this so list3 in unchanged, you can use the tuple() function to assign the example list as a tuple to list3. Since tuples are immutable, they cannot be dynamically altered like lists can. This will result in list3 being unchanged.

However, you may say "it's technically different because it's a TUPLE and not a list." You can easily use the list() function to turn list3 back into a list from a tuple.

In [51]:
```
# Alternatively, you can do the above steps in one line:

list1 = [1, 2, 99]
list2 = list1
list3 = list(tuple(list2))
list1 = list1.remove(1)

print(list3)
```

```
[1, 2, 99]
```

Alternatively, you can do this in one line with: list3 = list(tuple(list2)) and it will produce [1, 2, 99].

I believe this works because: Since when the list becomes a tuple, it is no longer mutable and is treated as a separate object and transforming it back into a list still treats it as a separate object. These are just my thoughts--I am curious to see if I'm on the right track here.

Question 7

In [52]:
```
# CHAPTER 7, QUESTION 7

# Consider:

ListA = [1, 2, 3, 4, 5]
ListB = ListA
ListA[2] = 10

# What is the value of ListB[2]?

print(ListB[2])
```

```
10
```

The value of ListB[2] is 10. This is because lists are immutable, meaning they can be dynamically changed. If you assign a list to a variable (or multiple variables), that variable's value will dynamically change with the list.

Question 19

In [53]:
```
# CHAPTER 7, QUESTION 19

# If a tuple is immutable, why are we able to modify x = [1, (2, 3), 4] into x = [1, (5, 6), 4] ?
# Are we changing the tuple or changing the list?

x = [1, (2, 3), 4]

# We are changing the list, NOT the tuple.

# Running either of the two lines below will produce an error.

# x[1][0] = 5 # or
# x[1][1] = 6

# However, if we attempt to change the values at the list level vs directly in the tuple:

x[1] = (5, 6)
print(x)
```

```
[1, (5, 6), 4]
```

We are changing the list, NOT the tuple. If we attempt to directly change the first value of the tuple, it will produce an error. Running either of those two commented lines will produce an error. If we attempt to change the values at the list level, it will successfully change from (2, 3) to (5, 6). We are changing the value of x[1] in the list and not directly interacting with the current tuple value.

Question 23

In [54]:
```
# CHAPTER 7, QUESTION 23

# Given the list L = [1, 3, 5, 7, 9], use slicing to create a new list without the value 3. That is, L2 = [

L = [1, 3, 5, 7, 9]

L2 = L[:1] + L[2:]

print("L: ", L, "\nL2: ", L2)
```

```
L:  [1, 3, 5, 7, 9]
L2:  [1, 5, 7, 9]
```

To do this, I can slice the list into two pieces. The first one, L[:1], to include up to but not including index 1 (which is the value 3). The second one, L[2:], includes values starting at index 2 (which is the value 5). Then we can concatenate the two lists and assign to L2