Assignment 1: Part 2 of 3

CODIO Project Resources

All project source files are found via Codio environment. This structure of this document for each question is as follows:

- Question description
- Confirmation of understanding of the task at hand
- Screenshot of the Codio solution
- Table containing the test and test results
- Presentation of alternate solutions (if any)
- Further considerations (if any) of the solution

Assignment Requirements

“Submit a single pdf file with all the deliverables mentioned above”

Question 1

Code a function called `first_div_16`

- ACCEPT two positive integers, *n1* and *n2*, as inputs
- RETURN the first number in range(*n1, n2*) that is divisible by 16.
- **However**, if no number in the range is divisible by 16, RETURN 0.

Requirements Understanding

The requirement is to determine the *first* number between two numbers that is wholly divisible by 16. *Wholly divisible* I take to mean “no remainder”. Since we accept INTEGER inputs, the solution utilises the *modulo* operator.

Python Code Solution

```
1  def first_div_16(n1, n2):
2    # The first requirement is to accept two INTEGER values
3    # So let's make sure our input data types are correct.
4    # We COULD coerce float data types into int values by
5    # using type cast... BUT that does not adhere to the requirements
6    if type(n1) is not int and type(n2) is not int:
7      return 0
8
9    # The requirement is that the numbers are positive
10   if n1 <= 0 or n2 <= 0:
11     return 0
12
13   # input data could be n1 == n2. If so, we simply test the first one
14   if n1 == n2:
15     if n1 % 16 == 0:
16       return n1
17
18   # Now we test each number within the range between n1 and n2.
19   # We use the "range" function and specify "+1" for the second param.
20   # This is because the "range" function not include the end number
21   # by default.
22   for x in range(n1, n2 + 1):
23     if x % 16 == 0:
24       return x
25
26   # We've already checked the given range, so what remains at this
27   # point is a "false" condition
28   return 0
```

Figure 1 Code for solution #1

Tests and Results

```
39  # test data
40  values = [(9,56), (16,16), (15, 16),
41            (-209, 16), (0, 9),
42            (14.01, 32.0),
43            ("ant", "hill")]
44  # test counter
45  test = 1
46
47  # loop through each test
48  for val in values:
49      print(f"Test {test}: {val} = " +
50            str(first_div_16(val[0], val[1])))
51      test += 1
```

```
Test 1: (9, 56) = 16
Test 2: (16, 16) = 16
Test 3: (15, 16) = 16
Test 4: (-209, 16) = 0
Test 5: (0, 9) = 0
Test 6: (14.01, 32.0) = 0
Test 7: ('ant', 'hill') = 0
```

Alternate Solutions

1. **Alternate Solution: *Replace* `range()` *with a* `while` *loop***

   In this solution, we replace the use of `range()` with a manual `while` loop and continually increment `n1` until it is either equal to `n2` or it is divisible by 16 at which point we exit the loop. Before we exit the loop, we print the output value if the number is divisible by 16.

   This is **not** recommended because it is not as efficient when writing the solution code because of the increased number of source lines.

2. **Alternate Solution: *Check* `n1` > `n2`**

   In this solution, we check whether `n1` > `n2`. If so, we swap around the two values. The solution continues to work as described. This allows callers to not concern themselves with which order the two inputs must be provided. Doing so would allow code such as `first_div_16(20, 1)` and `first_div_16(1, 20)` to work interchangeably.

   This **is** a recommended change to the existing solution but may be overkill since the requirements did not ask for such checks to be delivered.

Considerations

- Assume that `n1` < `n2` will always be true therefore `range`() is suitable since its default step value is “1”.

- `float` types can be converted to `int` types; however, this is does not meet the (client's) requirements "two positive **integers**".

Question 2

Code a function called `halve_to_2`

- ACCEPT one numeric input.
- If the number <= 0, RETURN -1.
- If the number > 0, divide that integer over-and-over by 2 until it becomes smaller than 2.
- RETURN that smaller-than-2 number, e.g., input of 4 will yield 1 (4->2->1), 5 yields 1.25 (5->2.5->1.25) etc.

Requirements Understanding

The requirement is to find the first number smaller than 2, that is either an `integer` or `float` value type. To find this number, the solution must continually divide a given input by 2.

Python Code Solution

```python
def halve_to_2(n1):
  # Do a type check to ensure we can indeed use the "/"
  # operator on the input. This is valid for float and int
  # types, but would fail for string
  isint = type(n1) is int
  isfloat = type(n1) is float
  isbool = type(n1) is bool

  if not isint and not isfloat and not isbool:
    return -1

  # Special condition for booleans, since the output
  # would show a a boolean True or False, however we
  # want to show it as an integer value
  if isbool:
    n1 = int(n1)

  # No need to run the algorithm if the number does not meet
  # the specified requirement of "If number > 0..."
  if n1 <= 0:
    return -1

  # We use the "while n1 >= 2" because the requirement is
  # "...until the result becomes smaller than 2". If we simply
  # used the ">" operator, our loop would stop at the value
  # "2" itself and we would not meet the requirements for the
  # solution. For example, 16 / 2= 8 / 2 = 4 / 2 = 2
  #
  # We use normal "/" operator because we know that both inputs
  # are integers and don't really need to use "//" operator
  while n1 >= 2:
    n1 = n1 / 2

  return n1
```

Figure 2 Code for solution #2

Tests and Results

```
36  # test values
37  values = [96, 15.67,
38            "Test", True,
39            -120, 0,
40            1, 2]
41
42  # test counter
43  test = 1
44
45  #iterate each test
46  for val in values:
47    print(f"Test {test}: {val} = " +
48          str(halve_to_2(val)))
49    test += 1
```

```
Test 1: 96 = 1.5
Test 2: 15.67 = 1.95875
Test 3: Test = -1
Test 4: True = 1
Test 5: -120 = -1
Test 6: 0 = -1
Test 7: 1 = 1
Test 8: 2 = 1.0
```

Alternate Solutions

No additional solutions determined to be better.

Considerations

- the *input data type*. This checked is executed to prevent run-time exceptions from occurring at the caller's side. We want to write reliable, stable code and we know that string data types do not support the "/" operator.
- Use of *modulo 2.* This will not work because `4 % 2 = 0` and does not meet the requirements.
- Hash tables to quickly lookup the smallest number for a given input. Downside is increased memory footprint, increased algorithmic complexity.

Question 3

Code a function called `string_expansion`.

- ACCEPT a non-empty string as input
- RETURN a string that contains every other character, 2n+2 times, where n is the original index of the letter. e.g., Input of “Hello” should result in "HHlllllllooooooooooo". Input of “ROBErt” should result in "RRBBBBBBrrrrrrrrrr".

Requirements Understanding

The solution prints the first and thereafter each second character. The selected character is replicated based on its index position within the string plus 2 additional characters.

Python Code Solution

```
def string_expression(p1):
  # make sure we are using a string type...
  if type(p1) is not str:
    return ""

  # make sure the string has content inside it
  # the "len" function will test for the content
  # size of the string value
  if len(p1) == 0:
    return ""

  # We use the concept of string splitting and "strides"
  # to pick out every 2nd character. Here we specify the
  # starting point in our string ("0") and instruct Python
  # to extract every second character ("::2")
  extracted = p1[0::2]

  result = ""
  index = 0

  # Now we loop through each character extracted and replicate
  # it according to the solution requirements which is
  # its index in the string + 2
  # We use (2 * index) to correctly position ourselves according
  # to the "every second character" requirement. Since strings
  # are indexed starting from zero, the first character (2 * index) =
  # (2 * 0)
  for char in extracted:
    # The "*" operator can be used to duplicate a string
    result= result + (char * ((2 * index) + 2))

    # Now we skip ahead by two characters
    index+=2

  return result
```

Figure 3 Code for solution #3

Tests and Results

```
# test values
values = ["ROBErt", "Hello", "Sir", "A", "",
          12345, ['a', 'string', 9.0]]

# test counter
test = 1

# perform each test
for val in values:
    print(f"Test {test}: {val} = " +
          str(string_expression(val)))
    test += 1
```

```
Test 1: ROBErt = RRBBBBBBrrrrrrrrrr
Test 2: Hello = HHlllllloooooooooo
Test 3: Sir = SSrrrrrr
Test 4: A = AA
Test 5:  =
Test 6: 12345 =
Test 7: ['a', 'string', 9.0] =
```

Alternate Solutions

1. Alternate Solution: Using `for` loops

For this solution, we would replace the string slicing routing ("`extracted = p1[0::2]`") with a `for` loop.

```
result = ""
index = 0
for char in p1:
  if (index + 1) % 2 == 0:
    index += 1
    pass

  result= result + (char * ((2 * index) + 2))
  index += 1
```

Considerations

No further considerations for this solution.

Question 4

Code a function called `item_count_from_index`.

- ACCEPT two inputs, a list and an integer-index
- RETURN a count (number) of how many times the item at that index appears in the list.
- **However**, if the integer-index is out of bounds for the list RETURN the empty string (“”) (e.g., list of 3 items, index of 5 is out of bounds)

Requirements Understanding

The solution is to return the number of occurrences of an element within a list.

Python Code Solution

```
def item_count_from_index(il, idx=0):
  # do a type check to ensure we are dealing with a list type
  if type(il) is not list:
    return ""

  # we check for an empty list
  if len(il) == 0:
    return ""

  # ensure that 'idx' actually has a value
  if type(idx) is None:
    return ""

  # ensure the supplied index is within bounds of the list
  # We use len(li) - 1 because lists in Python are zero-based
  # and we don't want an idx value that is the same as the
  # length of the list as this would create a run-time
  # exception. So, zero-based lists have an end index of
  # len(<list>) - 1
  if idx < 0 or idx > len(il) - 1:
    return ""

  # Now we can safely retrieve the element in the dataset
  # at the specified index
  item = il[idx]

  # The Python function count() is used to count the occurences
  # of a given value in the list
  count = il.count(item)

  return count
```

Figure 4 Code for solution #4

Tests and Results

```
33  # test values. We use tuples to capture the test data
34  # We use "None" to represent the absence of data NOT
35  # a NULL value which is itself the representation
36  # of a value
37  values = [(2,None),
38            ("string",None),
39            (False,None),
40            ([],None),
41            ([1,4,7,2,4,4,2],5),
42            ([1,4,7,2,4,4,2],60),
43            ([1,4,7,2,4,4,2], 2),
44            (["shopping", "mall"], 1)]
45  test = 1
46  for val in values:
47    count = item_count_from_index(val[0], val[1])
48    if count != "":
49      elm = val[0][val[1]]
50      print(f"Test {test}: {val} = " +
51          f"Found '{elm}', {count} times")
52    else:
53      print(f"Test {test}: {val} = Element not found")
54
55    test +=1
```

```
Test 1: (2, None) = Element not found
Test 2: ('string', None) = Element not found
Test 3: (False, None) = Element not found
Test 4: ([], None) = Element not found
Test 5: ([1, 4, 7, 2, 4, 4, 2], 5) = Found '4', 3 times
Test 6: ([1, 4, 7, 2, 4, 4, 2], 60) = Element not found
Test 7: ([1, 4, 7, 2, 4, 4, 2], 2) = Found '7', 1 times
Test 8: (['shopping', 'mall'], 1) = Found 'mall', 1 times
```

Alternate Solutions

1. Alternate Solution: Using `for` loops

Replace the use of Python's `count()` function with a for loop to iterate through the list comparing each element to the one found at the specified index.

This approach is **not** recommended because it is essentially a linear search algorithm and will not perform optimally if the list need to be searched multiple times.

Considerations

- To ensure the code is reliable and does not fail, we enforce a type-check for a `list` type
- For the unit test, we wanted to display the actual element being searched for and not `print()` statements into the function. This is to maintain a single-responsibility software design pattern.

Question 5

Code a function called `length_times_largest`.

- ACCEPT a list as input
- RETURN the length of the list times the largest integer (not float) in the list.
- **However**, if the list does not contain an integer, RETURN the empty string (“”).

Requirements Understanding

The solution requirement is to return the size of the list multiplied by largest integer element value found within. Based on “if the list does not contain **an** integer” (at least one integer), I take this to mean that the solution will find the largest integer in the list even if other value types are found in the list.

Python Code Solution

```
1  def length_times_largest(li):
2    # Ensure we are working with a list type
3    if type(li) is not list:
4      return ""
5
6    # Let's check to make sure the list contains some values
7    if len(li) == 0:
8      return ""
9
10   # We use a copy of the list because we do not want to
11   # modify the list passed to us by callers of this method
12   # This is a side-effect that we must either publish or prevent
13   temp = li.copy()
14
15   # We now enumerate all elements in the list. Since we want
16   # to make use of the index of the enumerated element, we
17   # use Python's "enumerate" function that allows us to
18   # request the item's index as well. With the returned
19   # index, we can then retrieve the element and test whether
20   # or not it is an integer (please see Requirements Understanding)
21   # for why this is done
22
23
24   for idx, x in enumerate(temp):
25     # if the element is not an int type, we simply replace its value
26     # with zero. This will make the call later on to "max(li)" far
27     # more simpler
28     if type(x) is not int:
29         temp[idx] = 0
30
31   # Use built-in function to calculate the max item value
32   max_val = max(temp)
33
34   return len(temp) * max_val
```

Figure 5 Code for solution #5

Tests and Results

```python
# test values
values = [90981, "test", 1.056,
          [],
          [1,2,3.04],
          ["hi", "there"],
          [2, 100, -65, 6.09, 8],
          [2.06, 'a', 'str', 50, -100],
          [-10, 16, 2, 45]]

# test counter
test = 1

# perform all tests
for val in values:
  largest = length_times_largest(val)
  if largest == "":
    print(f"Test {test}: {val} = " +
          str(largest))
  else:
    print(f"Test {test}: {val} " +
          f"({len(val)} items)= " + str(largest))
  test += 1
```

```
Test 1: 90981 =
Test 2: test =
Test 3: 1.056 =
Test 4: [] =
Test 5: [1, 2, 3.04] (3 items)= 6
Test 6: ['hi', 'there'] (2 items)= 0
Test 7: [2, 100, -65, 6.09, 8] (5 items)= 500
Test 8: [2.06, 'a', 'str', 50, -100] (5 items)= 250
Test 9: [-10, 16, 2, 45] (4 items)= 180
```

Alternate Solutions

No better solutions were identified.

Considerations

- We leverage the `enumerate()` function because we don't want the effort of maintaining a separate index counter.
- Applying `max()` over a list with any data type inside results in a run-time exception. Therefore, we *must* iterate and validate the type of entries in the list.