Lesson 1

January 15, 2025

1 Python 110, Lesson 1

1.1 Introduction to collections

A **collection** is a generic term for a container data type, which can hold multiple objects. A **sequence** is a subset of a container that maintains order. All sequences are collections but not all collections are sequences.

Sequences

Primary sequence types: * Strings (`str`) * Lists (`list`) * Tuples (`tuple`) * Ranges (`range`)

Strings are different fromt he rest in that the individual "parts" of a string don't actually exist as an individual part with a separate identity. In the list `['a', 'b', 'c']`, each string is an object. In the string`'abc'`, the characters are just part of the string.

Other collections * Dictionaries (`dict`) * Sets (`set`) * Frozen sets (`frozenset`)

1.1.1 Sequences

Lists **Lists represent an ordered collection f objects, characterized by zero-based indexing and mutability.** Each element of a list can be any data type, even another list. To access a specific element, you use its index, with a positive index (starting at 0 for the initial element) counting from the left or beginning and a negative index (starting at -1 for the final element). Attempting to access an index outside of the list's range results in an `IndexError`:

[10]:
```
lst = [1, 2, 3, 4]
lst[4]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[10], line 2
      1 lst = [1, 2, 3, 4]
----> 2 lst[4]

IndexError: list index out of range
```

A defining characterstic of lists is mutability. Element reassignmnet notation accesses the element on the left of the assignment operator and has a new value on the right of the assignment operator, like this: `lst[0] = 5`. This is considered a **destructive action**. It mutates the list.

In the Python REPL or a script, increment values of the remaining numbers in the numbers list

```
[ ]: numbers = [2, 2, 3, 4]
for idx, number in enumerate(numbers):
    if idx != 0:
        numbers[idx] += 1
print(numbers)
```

Tuples **Tuples are ordered, immutable collections**. Tuples are references/accessed by their index just like lists, and attempting to access a list outside of the tuple's range still results in an `IndexError`. A tuple's elements cannot be modified, however, the entire tuple can be reassigned, like so:

```
[ ]: games_to_play = ('traveller', 'dolmenwood', 'deadlands')
print(games_to_play)
games_to_play = ('traveller', 'deadlands', 'coriolis')
print(games_to_play)
```

Strings **Strings are ordered, immutable collections of characters.** They can be indexed just like lists and tuples and can result in an `IndexError` just like lists and tuples. Like tuples, a string can't be mutated, but the variable can be reassigned to a new string object.

Ranges **Ranges represent sequences of ineger numbers and are immutable, arithmetic progressions of integers**. They are 'lazy': only computed as needed. Ranges can also be accessed by element and will result in an `IndexError` if incorrectly accessed. Ranges can be reassigned, but individual elemnts of a range cannot be (just like strings and tuples). The example below is an example of a range being lazy - printing the range or a variable pointing to the range will just display the entire range because python won't generate it unless it has to. Converting it to a list, calling the individual elements, or iterating through it will show each element.

```
[ ]: num = range(5, 10)
print(range(5, 10))
print(num)
print(len(num))
print(num[0], num[1], num[2], num[3], num[4])
print(list(num))
for i in num:
    print(i)
```

This makes ranges memory efficient: the only thing pyuthon needs to physically store in memory is the start, stop, step, and current/most recent value.

Operations on Sequences **Slicing retrieves a subset of elements from a saequence**. The syntax is `sequence[start:stop:step]`. Important reminders:

- `stop` is exclusive: it's the first number after `start` that is not included in the slice.
- `start`'s default is index 0 for positive `step` values and index -1 for negative `step` values.
- `step` indicates the interval between positions. Default is 1

- `step` -1 (like so: `sequence[::-1]` reverses th sequence.
- Slicing does not mutate. The retrieved or altered subset is a return value of the slicing operation
- If `start` is greater than `stop`, the result is an empty sequence
  - In the case of a *negative* ***`step`***, if `stop` is greater than `start`, the result is an empty sequence
- `lst[:]` creates a shallow copy. A shallow copy creates a copy of the sequence, but creates a reference to the objects inside the sequence. If there are variables in the sequence, the shallow copy will reference the them. If there are other sequences in the sequence, the shallow copy will reference them. Trace the shallow copy in the example below.

[ ]:
```
tup = (1, 2, 3, 4, 5)
tup[::-1]
new_tup = tup[::-1]
print(tup)
print(new_tup)

lst = [1, 2, 3, 4, 5]
lst[::-1]
new_lst = lst[::-1]
print(lst)
print(new_lst)

# lst[:] creates a shallow copy
a = lst[:]
a.append(6)
lst.append(0)
print(a)
print(lst)

# to show that it's a shallow copy:
lst_to_append = [11, 21, 31, 41]
print(f'\nnew_lst is {new_lst}')
print(f'\nlst_to_append is {lst_to_append}')
new_lst.append(lst_to_append)
print(f'\nNow that we\'ve appended, new_lst is {new_lst}')
sliced_shallow_copy_of_new_lst = new_lst[:]
print(f'\nWe created a sliced shallow copy of new_lst:␣
 ↪{sliced_shallow_copy_of_new_lst}')
lst_to_append.append(51)
print(f'\nNow we\'ve appended 51 to lst_to_append: {lst_to_append}')
print(f'\nnew_lst is {new_lst}')
print(f'\nsliced_shallow_copy_of_new_lst is {sliced_shallow_copy_of_new_lst}')
new_lst[0]='here is a change to new_lst'
sliced_shallow_copy_of_new_lst[1]='here is a change to␣
 ↪sliced_shallow_copy_of_new_lst'
print(f'\nReassigning new_lst[0]: {new_lst}')
```

```
print(f'\nReassigning sliced_shallow_copy_of_new_lst[1]:␣
  ↪{sliced_shallow_copy_of_new_lst}')
lst_to_append[0]='here is a change to lst_to_append'
print(f'\nnew_lst after reassigning lst_to_append[0]: {new_lst}')
print(f'\nsliced_shallow_copy_of_new_lst after reassigning after reassigning␣
  ↪lst_to_append[0]: {sliced_shallow_copy_of_new_lst}')
```

`len` returns the length of a sequence - the number of elements.

All sequences are iterables. Python can iterate over them in a loop. A for loop is the most pythonic choice:

```
strang = 'purple'
for char in strang:
    print(character)
```

```
p
u
r
p
l
e
```

You can use while loops and a counter to get indices if for some reason you don't want to use enumerate (enumerate is better):

[ ]:
```
lst = ['apples', 'bananas', 'grapes']
i = 0
print('Using a while loop with a counter:')
while i < len(lst):
    print(f'item {i}: {lst[i]}')
    i += 1

print('\nUsing a for loop with a enumerate:')
for idx, item in enumerate(lst):
    print(f'item {idx}: {lst[idx]}')
```

The + operator concatenates sequences of the same type. This creates a new sequence rather than modifying the original.

Ranges cannot be concatenated and attempts will return an IndexError.

[ ]:
```
# These are three different objects
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print(f'The id of {a} is {id(a)},\nThe id of {b} is {id(b)},\nThe id of {c} is␣
  ↪{id(c)}.')
```

`count`, `indexm` `min`, and `max` are also available for all sequences. `count` returns the number of occurances of a value in the sequence:

```
lst = [1, 2, 2, 3, 4, 4, 4]
lst.count(4)
3
```

`index` returns the index of the *first occurence* of a value in the sequence. If it's not there, it raises a `ValueError`.

```
lst.index(4)
4
```

For lists and tuples, executing min and max on numeric values returns the smallest and largest values. If there are strings, the highest character by unicode value is returned. With mixed data types (string and int) a `TypeError` is raised.

Conversion to Lists and Tuples Convert to lists using `list()`. Passing a string in this way will make each character an element in the list. Passing a tuple to `list` retains the order of the tuple:

```
>>> my_tup = (0, 1, 2, 3)
>>> list(my_tup)
[0, 1, 2, 3]
```

Convertring a range to a list will generate each number in the range:

```
>>> r = range(5)
>>> list(r)
[0, 1, 2, 3, 4]
```

Using `tuple()` and `str()` works similarly to `list()` but for conversion to tuples and strings. The `join` method with a generator expression is frequently better. The syntax goes: text.join(iterable), so that

```
my_list = [192,168,0,1]
a = ".".join([str(element) for element in my_list])
print(a)

192.168.0.1
```

1.1.2 Dictionaries, Sets, and Frozen Sets

Dictionaries - Basic Operations While these are iterable, they aren't sequences.

Dictionaries are collections of key-value pairs. They do maintain the order of insertion which makes them a lot like sequences. Importantly, they are mutable and you have to be careful passing the values.

- Access values by key: `dict[key]`
  - Key not present = `KeyError`
- Delete key-value pair: `del dict[key]`
  - Key not present = `KeyError`
- Does a key exist?
  - `key in dict`
  - `key not in dict`

Dictionaries - Common methods

- Shallow copy of dictionary = `dict.copy()`
- Fetch value based on key or return default value: 'dict.get('key', 'default_value')
  - Doesn't set the value by default, just returns the value.
- To set a value for a key if the key does not exist but to leave the value alone if it exists, use `dict.setdefault('key', 'default')`.
  - IF key-value pair exists, does not mutate, does return existing value
  - If key-value pair does not exist, creates them using arguments, and returns new value
- To remove a k-value pair and return the value, use `dict.pop(key)`.
  - IF the key doesn't exist, raises `KeyError`
  - Can add a default value to return if the key-value pair doesn't exist, like this: `dict.pop('city', 'princeton')` would delete the k/v pair if `city` exists and then return the value of `city`, but if `city` doesn't exist, it would return `princeton`.
- If you need both the key and the value, use `dict.popitem()`, which removes the last k-v pair and returns it as a tuple
  - If the dictionary is empty, this raises a 'KeyError

Merging Dictionaries

- To merge two dictionaries into one, use the update method: `dict.update(dict2)`
  - If keys overlap, the second dictionaries will supercede the first (dict2's k-v pairs will replace dict1's)
  - Mutates dict1
- To leave the original dictionaries alone, use the merge operator: `|`. This returns a new dictionary like this: `dict3 = dict1 | dict2`
- A shorter syntax for `update` is `|=`. This mutates the first dictionary (*left of the operator*).

Conversion to Dictionaries

- Key-Value pairs of other data types can be converted to dictionaries using `dict(pairs)`

1.1.3 Sets

- Does a value exist in a set?
  - `value in s`
  - `value not in s`
- To say that `a is a subset of b` means that all elements of `set a` are in `set b`.
  - `set_a.issubset(set_a)` (returns boolean)
  - `set_a <= set_b` (returns boolean)
- To say that `b is a superset of a` means that all elements of `set a` are in `set b`. It's the same relationship as in subsets but from the perspective of the other set.
  - `set_b.issuperset(set_a)` (returns boolean)
  - `set_a >= set_b` (returns boolean)
- Important note! The above operators (`<=` and `>=`) will also return true for equal subsets and supersets. To return true for only `proper subsets` and `proper supersets`, use `<` and `>` instead.

Set Operators

- The **union** combines elements of the two sets. Note that sets by definition contain only unique items.
  - `set1.union(set2)` and `sert1 | set2`= returns a value, does not mutate
- The **intersection** finds the elements in common between two sets. Does nto mutate, only returns a value
  - `set1 & set2` or `set1.intersection(set2)`
- The **difference operator** returns what's in one set but not the other. Also doesn't mutate, just returns a value.
  - `set1 - set2` or `set1.difference(set2)`
- If two sets have nothing in common, they're considered `disjoint`. This returns a boolean
  - `set1.isdisjoint(set2)`

Set Methods

- To **copy** a set use the copy method
  - `set1.copy()` returns a new set that must be assigned to a variable.
- `set1.add(item)` adds an element to an existing set. Doesn't do anything if the set already has that member.
- `set1.remove(item)` raises a `KeyError` if the item isn't in the set
- `set1.discard(item)` removes an item but raises no error if the item doesn't exist
- `set1.clear()` discards all elements and leaves an empty set in place
- `set1.pop()` removes and returns an arbitrary element from the set. Raises a `KeyError` if the set is empty

1.2 Unpacking Iterables in Python

The unary operator `*` "unpacks" iterables:

```
my_list = ['a', 'b', 'c']
my_tup = (1, 2, 3)
joined_tuple = (*my_list, *my_tup)
joined_list = [*my_list, *my_tup]

>>joined_tuple
('a', 'b', 'c', 1, 2, 3)
>>joined_list
['a', 'b', 'c', 1, 2, 3]
```

It can calso be used to cleanly unpack variables into function arguments:

```
def this_function(num1, num2, num3):
    pass

numbers = [1, 2, 3]

this_function(*numbers)
```

1.2.1 Unary Operator and Dictionaries

Dictionaries use the double-asterisk unary operator: **.

```
>>> dict1 = {'a': 1, 'b': 2}
>>> dict2 = {'b': 3, 'd': 4}
>>> merged_dict = {**dict1, **dict2}
>>> merged_dict
{'a': 1, 'b': 3, 'd': 4}
```

1.3 Introduction to the PEDAC Process

//General notes here

Walk through Understand the [**P**]roblem for this:

```
"""
PROBLEM:

Given a string, write a function `palindrome_substrings` which returns
all the palindromic substrings of the string that are 2 or more characters
long. Palindrome detection
should be case-sensitive.

# Test cases:

# Comments show expected return values
palindrome_substrings("abcddcbA")    # ["bcddcb", "cddc", "dd"]
palindrome_substrings("palindrome") # []
palindrome_substrings("")           # []
palindrome_substrings("repaper")    # ['repaper', 'epape', 'pap']
palindrome_substrings("supercalifragilisticexpialidocious") # ["ili"]
```

Understand the problem:

Input: string (explicit in problem) Output: list (implicit in test cases)

function signature: palindrome_substring(string)

! notes * Palindromes are at least two characters long * Case matters * Characters can belong to multiuple substrings * The entire string can be a palindrome and should be returned * Substrings can be even or odd

1.4 PEDAC Guide Practice

1.4.1 Leftover Blocks

Step 1: Understand the Problem: You have a number of building blocks that can be used to build a valid structure. There are certain rules about what determines a valid structure:

- The building blocks are cubes.
- The structure is built in layers.
- The top layer is a single block.
- A block in an upper layer must be supported by four blocks in a lower layer.

- A block in a lower layer can support more than one block in an upper layer.
- You cannot leave gaps between blocks.

Write a program that, given the number of available blocks, calculates the number of blocks left over after building the tallest possible valid structure.

- Make notes of your mental model for the problem, including:
  - inputs and outputs.
  - explicit and implicit rules.
- Write a list of clarifying questions for anything that isn't clear.

Make notes of your mental model: Inputs: number of blocks (integer) Outputs: number of blocks remaining (integer)

Write a list of clarifying questions for amything that isn't clear: 1. Are the blocks the same size 2. Is there a certain amount of a lower block that an upper block has to sit on to be considered "supported"? For example, if an upper block rests at the center of four blocks lying in a plane, with the upper block supported by four quarters, is that good enough? 3. How far down does the suppport requirement go? Is this an infinite level of bases?

Step 2: Examples and Test Cases Regarding your initial mental model and questions from Step 1, make some notes about the test cases. Do the test cases confirm or refute different elements of your original analysis and mental model? Do they answer any of the questions that you had, or do they perhaps raise further questions?

```
print(calculate_leftover_blocks(0) == 0)  # True
print(calculate_leftover_blocks(1) == 0)  # True
print(calculate_leftover_blocks(2) == 1)  # True
print(calculate_leftover_blocks(4) == 3)  # True
print(calculate_leftover_blocks(5) == 0)  # True
print(calculate_leftover_blocks(6) == 1)  # True
print(calculate_leftover_blocks(14) == 0) # True
```

It looks like 5 blocks make 2 layers and 14 blocks make 3 layers, so

```
   |_|      1 (1x1)
  |_|_|     4 (2x2)
 |_|_|_|    9 (3x3)
|_|_|_|_|  16 (4x4)
```

Step 3: Data Structures Ok, so the output is an integer. What intermediate structures do I need? I could precalculate the number of blocks for specific levels, but that doesn't seem worthwhile. Honestly, I think there may be nothing significant here.

Step 4: Algorithm

- receive a number of blocks
- initialize a level counter
- initialize a leftover block counter, assign it to the number of blocks passed as an argument
- for each level, the counter will equal the number of sides - square that to tell how many blocks are required

- check to see if there are enough blocks to build the next level
  - if yes, reduce the leftover block counter by the square, then increment the level counter
  - if no, return the leftover number of blocks

[ ]:
```python
def calculate_leftover_blocks(blocks):
    leftover_blocks = blocks
    current_level = 0

    while True:
        current_level_blocks = (current_level + 1) ** 2
        if current_level_blocks <= leftover_blocks:
            leftover_blocks -= current_level_blocks
            current_level += 1
        else:
            return leftover_blocks

print(calculate_leftover_blocks(0) == 0)  # True
print(calculate_leftover_blocks(1) == 0)  # True
print(calculate_leftover_blocks(2) == 1)  # True
print(calculate_leftover_blocks(4) == 3)  # True
print(calculate_leftover_blocks(5) == 0)  # True
print(calculate_leftover_blocks(6) == 1)  # True
print(calculate_leftover_blocks(14) == 0) # True
```

1.4.2 Sort by Most Adjacent Consonants

Step 1: Understand the Problem Given a list of strings, sort the list based on the highest number of adjacent consonants a string contains and return the sorted list. If two strings contain the same highest number of adjacent consonants, they should retain their original order in relation to each other. Consonants are considered adjacent if they are next to each other in the same word or if there is a space between two consonants in adjacent words.

Tasks You are provided with the problem description above. Your tasks for this step are:

- Make notes of your mental model for the problem, including:
  - inputs and outputs.
  - explicit and implicit rules.
- Write a list of clarifying questions for anything that isn't clear.
- I assume that it means a "chain" of adjacent consonants, like bcdght would = 5, but bdaght would = 3
  - Consonants are still adjacent if there is a space between them in adjacent words
- The list of strings could include strings with spaces, can it also include punctuation and special characters?
- Input a list of strings, output a sorted list of the same strings
- If that value is the same between two strings, they should retain their original order relative to index 0

Step 2: Examples and test cases

```
my_list = ['aa', 'baa', 'ccaa', 'dddaa']
print(sort_by_consonant_count(my_list))
# ['dddaa', 'ccaa', 'aa', 'baa']

my_list = ['can can', 'toucan', 'batman', 'salt pan']
print(sort_by_consonant_count(my_list))
# ['salt pan', 'can can', 'batman', 'toucan']

my_list = ['bar', 'car', 'far', 'jar']
print(sort_by_consonant_count(my_list))
# ['bar', 'car', 'far', 'jar']

my_list = ['day', 'week', 'month', 'year']
print(sort_by_consonant_count(my_list))
# ['month', 'day', 'week', 'year']

my_list = ['xxxa', 'xxxx', 'xxxb']
print(sort_by_consonant_count(my_list))
# ['xxxx', 'xxxb', 'xxxa']
```

This reinforces some of my assumptions: there can be strings with spaces, strings can be multiple words.

There's also some new details that should have been obvious: we aren't counting consonants, just adjacent consonants: `bac` has a value of `0`

Step 3: Data Structures For this step, with reference to your analysis form the two peceding steps, make some notes regarding whether you need to use any particular data structures in your solution.

At this phase, I'm tempted to shift to a dictionary as an intermediate step to count values before reassigning to a list for return.

Step 4: Algorithm: For this step, use your analysis of the problem so far to write out a high-level algorithm that solves the problem at an abstract level. Avoid too much implementation detail at this stage.

- create an empty dictionary called values
- for item in list:
  - values[item] = count_adjacent_consonants(item)
- check for duplicate values, if there are duplicates, order them by index in list (add 1 to the dictionary value for lower-indexed items)
- insert the keys back into list
- return list

ok, the algo to count adjacent consonants

- pre-first, let's make a list of vowels
- First, spaces don't matters, so let's remove them
- Then, loop through the characters

- set a current_counter to zero
- set a max_counter to zero
- if character not in vowel list, increment current_counter
- if character in vowel list, compare current counter to max counter and increment if necessary
- if end of string, return max counter

Step 5: IUmplement a solution in Code:

[ ]:
```
def count_adjacent_consonants(string):
    VOWELS = ('a',
              'e',
              'i',
              'o',
              'u'
             )
    eval_string = string.lower().replace(" ","")
    temp_string = ""
    max_counter = 0
    for char in eval_string:
        if char.isalpha() and char not in VOWELS:
            temp_string += char
        else:
            if len(temp_string) > max_counter:
                max_counter = len(temp_string)
            temp_string = ""

    if len(temp_string) > max_counter:
        max_counter = len(temp_string)

    return 0 if max_counter <= 1 else (max_counter)

def sort_by_consonant_count(lst):
    return sorted(lst, key=count_adjacent_consonants, reverse=True)

my_list = ['aa', 'baa', 'ccaa', 'dddaa']
print(sort_by_consonant_count(my_list))
# # ['dddaa', 'ccaa', 'aa', 'baa']

my_list = ['can can', 'toucan', 'batman', 'salt pan']
print(sort_by_consonant_count(my_list))
# ['salt pan', 'can can', 'batman', 'toucan']

my_list = ['bar', 'car', 'far', 'jar']
print(sort_by_consonant_count(my_list))
# # ['bar', 'car', 'far', 'jar']

my_list = ['day', 'week', 'month', 'year']
```

```
print(sort_by_consonant_count(my_list))
# # ['month', 'day', 'week', 'year']

my_list = ['xxxa', 'xxxx', 'xxxb']
print(sort_by_consonant_count(my_list))
# # ['xxxx', 'xxxb', 'xxxa']
```

[ ]:
```
def count_adjacent_consonants(string):
    VOWELS = ('a', 'e', 'i', 'o', 'u')
    eval_string = string.lower().replace(" ", "")

    current_count = 0
    max_count = 0

    for char in eval_string:
        if char.isalpha() and char not in VOWELS:
            current_count += 1
            max_count = max(max_count, current_count)
        else:
            current_count = 0

    return 0 if max_count <= 1 else max_count

def sort_by_consonant_count(lst):
    return sorted(lst, key=count_adjacent_consonants, reverse=True)

my_list = ['aa', 'baa', 'ccaa', 'dddaa']
print(sort_by_consonant_count(my_list))
# # ['dddaa', 'ccaa', 'aa', 'baa']

my_list = ['can can', 'toucan', 'batman', 'salt pan']
print(sort_by_consonant_count(my_list))
# ['salt pan', 'can can', 'batman', 'toucan']

my_list = ['bar', 'car', 'far', 'jar']
print(sort_by_consonant_count(my_list))
# # ['bar', 'car', 'far', 'jar']

my_list = ['day', 'week', 'month', 'year']
print(sort_by_consonant_count(my_list))
# # ['month', 'day', 'week', 'year']

my_list = ['xxxa', 'xxxx', 'xxxb']
print(sort_by_consonant_count(my_list))
# # ['xxxx', 'xxxb', 'xxxa']
```

1.5 Selection and Transformation

In addition to **Iteration**, you can perform selection and transformation on collections. **Selection** is the process of picking some elements out of a collection depending on one or more criteria, like all odd numbers, or all squares, or all plurals. **Transformation** manipulates every element in a collection: appending `s` to all strings, squaring every number, etc.

```
produce = {
    'apple': 'Fruit',
    'carrot': 'Vegetable',
    'pear': 'Fruit',
    'broccoli': 'Vegetable',
}

print(select_fruit(produce))  # { apple: 'Fruit', pear: 'Fruit' }
```

How would I implement `select_fruit(dic)`?

“‘python

def select_fruit(dic): return_dict = {} for k, v in produce.items(): if v == ‘Fruit’: return_dict[k] = v return return_dict

[ ]:
```
#Test it here:

def select_fruit(dic):
    return_dict = {}
    for k, v in produce.items():
        if v == 'Fruit':
            return_dict[k] = v
    return return_dict

produce = {
    'apple': 'Fruit',
    'carrot': 'Vegetable',
    'pear': 'Fruit',
    'broccoli': 'Vegetable',
}

print(select_fruit(produce))

#Woohoo, it works!
```

[ ]:
```
# Implement a double_numbers function that mutates the list of numbers passed␣
 ↪as an argument

def double_numbers(numbers):
    for idx, num in enumerate(numbers):
        numbers[idx] *= 2
```

```
    return numbers

my_numbers = [1, 4, 3, 7, 2, 6]
print(double_numbers(my_numbers)) # [2, 8, 6, 14, 4, 12]
print(my_numbers)                 # [2, 8, 6, 14, 4, 12]
```

[ ]:
```
'''Here's an exercise for you: suppose we wanted to transform the numbers based␣
↪on their position in the list rather than their value? Try coding a solution␣
↪that doubles the numbers that have odd indexes:'''

def double_odd_indices_in_place(lst):
    for idx, num in enumerate(lst):
        if idx % 2 == 1:
            lst[idx] *= 2

    return lst

def double_odd_indices_with_new_list(lst):
    doubled_by_index = []
    for idx, num in enumerate(lst):
        if idx % 2 == 1:
            doubled_by_index.append(num * 2)
        else:
            doubled_by_index.append(num)
    return doubled_by_index
```

Continuing with the idea of building generic functions, let's replace our double_numbers function with a multiply function that can multiply the elements of the list by an arbitrary number. For instance:

```
my_numbers = [1, 4, 3, 7, 2, 6]
print(multiply(my_numbers, 3))  # [3, 12, 9, 21, 6, 18]
```

Try coding a function that lets you multiply every list element by a specified value. As with double_numbers, don't mutate the list, but return a new list instead.

[ ]:
```
def multiply(lst, multiplier):
    if not isinstance(multiplier, int) and not isinstance(multiplier, float):
        raise TypeError('The multiplier parameter accepts integers or floats␣
↪only')
    return [num * multiplier for num in lst]

my_numbers = [1, 4, 3, 7, 2, 6]
print(multiply(my_numbers, 3))  # [3, 12, 9, 21, 6, 18]
```

[ ]:
```
def replace_word_choice(sentence, old_word, new_word):
    """Replace a word in the provided sentence with a new one.
```

```
    :param sentence: str - a sentence to replace words in.
    :param old_word: str - word to replace.
    :param new_word: str - replacement word.
    :return: str - input sentence with new words in place of old words.
    """
    evaluation_list = sentence.split()
    for item in evaluation_list:

    ending_idx =

    return " ".join([new_word if word == old_word else word for word in␣
↪sentence.split()])

print(replace_word_choice("Animals are cool.", "cool", "awesome"))
```

1.6 Practice Problems

1.6.1 Practice Problem 1

Given the tuple:

```
fruits = ("apple", "banana", "cherry", "date", "banana")
```

How would you count the number of occurences of "banana" in the tuple?

Answer `fruits.count('banana')`

1.6.2 Practice Problem 2

Consider the set:

```
numbers = {1, 2, 3, 4, 5, 5, 6, 3}
print(len(numbers))
```

What is the set's length?

Answer 5 - duplicates don't count in sets.

1.6.3 Practice Problem 3

Given two sets:

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
```

How would you obtain a set that contains all the unique calues from both sets?

Answer `c = a | b` (correct - this would also work: `union_set = a.union(b)`

1.6.4 Practice Problem 4

Given the following code, what would the output be? Try to answer without running the code.:

```
names = ["Fred", "Barney", "Wilma", "Betty", "Pebbles", "Bambam"]
name_positions = {}
for index, name in enumerate(names):
    name_positions[name] = index
print(name_positions)
```

Answer {"Fred": 0, "Barney": 1, "Wilma": 2, "Betty": 3, "Pebbles": 4, "Bambam": 5}

1.6.5 Practice Problem 5

Calculate the total age given the following dictionary:

```
ages = {
    "Herman": 32,
    "Lily": 30,
    "Grandpa": 5843,
    "Eddie": 10,
    "Marilyn": 22,
    "Spot": 237,
}
```

Answer `sum_ages = sum([x for x in ages.values()])` (correct - this also works and is simpler: `total_age = sum(ages.values())`. I don't know how I missed that.)

1.6.6 Practice Problem 6

Determine the minimum age from the ages dictionary.

```
ages = {
    "Herman": 32,
    "Lily": 30,
    "Grandpa": 5843,
    "Eddie": 10,
    "Marilyn": 22,
    "Spot": 237,
}
```

Answer `min_age = min(ages.values())`

1.6.7 Practice Problem 7

What would the following code output?

```
words = ['ant', 'bear', 'cat']
selected_words = []
for word in words:
    if len(word) > 3:
```

```
        selected_words.append(word)

print(selected_words)
```

Answer `['bear']`

1.6.8 Practice Problem 8

Given the following string, create a dictionary that represents the frequency with which each letter occurs. The frequency count should be case-sensitive:

```
statement = "The Flintstones Rock"
```

The output should resemble the following, but your program may output the pairs in a different order:

```
# Pretty printed for clarity
{
    'T': 1,
    'h': 1,
    'e': 2,
    'F': 1,
    'l': 1,
    'i': 1,
    'n': 2,
    't': 2,
    's': 2,
    'o': 2,
    'R': 1,
    'c': 1,
    'k': 1
}
```

Answer

[ ]:
```
statement = "The Flintstones Rock"
#my solution
unique_characters = set(statement.replace(" ", ""))
frequency = {}
for item in unique_characters:
    frequency[item] = statement.count(item)
print(frequency)

#Launch School's solution
char_freq = {}
statement1 = statement.replace(' ', '')
for char in statement1:
    char_freq[char] = char_freq.get(char, 0) + 1

print(char_freq)
```

```
print(frequency == char_freq)
```

Practice Problem 9 What is the return value of the list comprehension below? Try to answer without running the code.

```
[num for num in [1, 2, 3] if num > 1]
```

Answer

```
[2, 3]
```

Practice Problem 10 What does the following code print and why?

```
dictionary = {'a': 'ant', 'b': 'bear'}
print(dictionary.popitem())
```

Answer

This returns ('b', 'bear') because popitem removes the last key-value pair as a tuple./

Practice Problem 11 What does the following code return? Try to answer without running the code.

```
lst = [1, 2, 3, 4, 5]
lst[:2]
```

Answer

This returns `[1, 2]`

[11]:
```
lst = [1, 2, 3, 4, 5]
lst[:2]
```

[11]:
```
[1, 2]
```

Practice Problem 12 What would be the output of the below code? Try to answer without running the code.

```
frozen = frozenset([1, 2, 3, 4, 5])
frozen.add(6)
print(frozen)
```

Answer

This returns `{1, 2, 3, 4, 5}` **WRONG**

```
AttributeError: 'frozenset' object has no attribute 'add'
```

[ ]:
```

```