Using the "split" string method from the preceding lesson, complete the get\_word function to return the {n}th word from a passed sentence. For example, get\_word("This is a lesson about lists", 4) should return "lesson", which is the 4th word in this sentence. Hint: remember that list indexes start at 0, not 1.

def get\_word(sentence, n):

    # Only proceed if n is positive

    if n > 0:

        words = sentence.split()

        # Only proceed if n is not more than the number of words

        if n <= len(words):

            return(words[n-1])

    return("")

print(get\_word("This is a lesson about lists", 4)) # Should print: lesson

print(get\_word("This is a lesson about lists", -4)) # Nothing

print(get\_word("Now we are cooking!", 1)) # Should print: Now

print(get\_word("Now we are cooking!", 5)) # Nothing

RunReset

Here is your output:

lesson

Now

Excellent! You're getting comfortable with string

conversions into lists. Now we are really cooking!

Lists Defined

Lists in Python are defined using square brackets, with the elements stored in the list separated by commas: **list = ["This", "is", "a", "list"]**. You can use the **len()** function to return the number of elements in a list: **len(list)** would return **4**. You can also use the **in** keyword to check if a list contains a certain element. If the element is present, it will return a True boolean. If the element is not found in the list, it will return False. For example, **"This" in list** would return True in our example. Similar to strings, lists can also use indexing to access specific elements in a list based on their position. You can access the first element in a list by doing **list[0]**, which would allow you to access the string **"This"**.

In Python, lists and strings are quite similar. They’re both examples of sequences of data. Sequences have similar properties, like (1) being able to iterate over them using **for loops**; (2) support indexing; (3) using the **len** function to find the length of the sequence; (4) using the plus operator **+** in order to concatenate; and (5) using the **in** keyword to check if the sequence contains a value. Understanding these concepts allows you to apply them to other sequence types as well.

The skip\_elements function returns a list containing every other element from an input list, starting with the first element. Complete this function to do that, using the for loop to iterate through the input list.

def skip\_elements(elements):

    # Initialize variables

    new\_list = []

    for x in range(len(elements)):

            new\_list.append(elements[i])

            i=i+2

    return new\_list

print(skip\_elements(["a", "b", "c", "d", "e", "f", "g"])) # Should be ['a', 'c', 'e', 'g']

print(skip\_elements(['Orange', 'Pineapple', 'Strawberry', 'Kiwi', 'Peach'])) # Should be ['Orange', 'Strawberry', 'Peach']

print(skip\_elements([])) # Should be []

    i=0

        if i< len(elements):

RunReset

Here is your output:

['a', 'c', 'e', 'g']

['Orange', 'Strawberry', 'Peach']

[]

You nailed it! Isn't all this list manipulation so much fun?

Modifying Lists

While lists and strings are both sequences, a big difference between them is that lists are mutable. This means that the contents of the list can be changed, unlike strings, which are immutable. You can add, remove, or modify elements in a list.

You can add elements to the end of a list using the **append** method. You call this method on a list using dot notation, and pass in the element to be added as a parameter. For example, **list.append("New data")** would add the string "New data" to the end of the list called list.

If you want to add an element to a list in a specific position, you can use the method **insert**. The method takes two parameters: the first specifies the index in the list, and the second is the element to be added to the list. So **list.insert(0, "New data")** would add the string "New data" to the front of the list. This wouldn't overwrite the existing element at the start of the list. It would just shift all the other elements by one. If you specify an index that’s larger than the length of the list, the element will simply be added to the end of the list.

You can remove elements from the list using the **remove** method. This method takes an element as a parameter, and removes the first occurrence of the element. If the element isn’t found in the list, you’ll get a **ValueError** error explaining that the element was not found in the list.

You can also remove elements from a list using the **pop** method. This method differs from the remove method in that it takes an index as a parameter, and returns the element that was removed. This can be useful if you don't know what the value is, but you know where it’s located. This can also be useful when you need to access the data and also want to remove it from the list.

Finally, you can change an element in a list by using indexing to overwrite the value stored at the specified index. For example, you can enter **list[0] = "Old data"** to overwrite the first element in a list with the new string "Old data".

Let's use tuples to store information about a file: its name, its type and its size in bytes. Fill in the gaps in this code to return the size in kilobytes (a kilobyte is 1024 bytes) up to 2 decimal places.

def file\_size(file\_info):

    name, Type, size= file\_info

    return("{:.2f}".format(size / 1024))

print(file\_size(('Class Assignment', 'docx', 17875))) # Should print 17.46

print(file\_size(('Notes', 'txt', 496))) # Should print 0.48

print(file\_size(('Program', 'py', 1239))) # Should print 1.21

Here is your output:

17.46

0.48

1.21

Well done! Aren't tuples handy to keep the information

nicely organized for when we need it?

Tuples

As we mentioned earlier, strings and lists are both examples of sequences. Strings are sequences of characters, and are immutable. Lists are sequences of elements of any data type, and are mutable. The third sequence type is the tuple. Tuples are like lists, since they can contain elements of any data type. But unlike lists, tuples are immutable. They’re specified using parentheses instead of square brackets.

You might be wondering why tuples are a thing, given how similar they are to lists. Tuples can be useful when we need to ensure that an element is in a certain position and will not change. Since lists are mutable, the order of the elements can be changed on us. Since the order of the elements in a tuple can't be changed, the position of the element in a tuple can have meaning. A good example of this is when a function returns multiple values. In this case, what gets returned is a tuple, with the return values as elements in the tuple. The order of the returned values is important, and a tuple ensures that the order isn’t going to change. Storing the elements of a tuple in separate variables is called unpacking. This allows you to take multiple returned values from a function and store each value in its own variable.

Iterating Over Lists Using Enumerate

When we covered *for* loops, we showed the example of iterating over a list. This lets you iterate over each element in the list, exposing the element to the for loop as a variable. But what if you want to access the elements in a list, along with the index of the element in question? You can do this using the **enumerate()** function. The enumerate() function takes a list as a parameter and returns a tuple for each element in the list. The first value of the tuple is the index and the second value is the element itself.

List Comprehensions

You can create lists from sequences using a for loop, but there’s a more streamlined way to do this: **list comprehension**. List comprehensions allow you to create a new list from a sequence or a range in a single line.

For example, **[ x\*2 for x in range(1,11) ]** is a simple list comprehension. This would iterate over the range 1 to 10, and multiply each element in the range by 2. This would result in a list of the multiples of 2, from 2 to 20.

You can also use conditionals with list comprehensions to build even more complex and powerful statements. You can do this by appending an if statement to the end of the comprehension. For example, **[ x for x in range(1,101) if x % 10 == 0 ]**would generate a list containing all the integers divisible by 10 from 1 to 100. The if statement we added here evaluates each value in the range from 1 to 100 to check if it’s evenly divisible by 10. If it is, it gets added to the list.

List comprehensions can be really powerful, but they can also be super complex, resulting in code that’s hard to read. Be careful when using them, since it might make it more difficult for someone else looking at your code to easily understand what the code is doing.

## Lists and Tuples Operations Cheat Sheet

## Lists and Tuples Operations Cheat Sheet

Lists and tuples are both sequences, so they share a number of sequence operations. But, because lists are mutable, there are also a number of methods specific just to lists. This cheat sheet gives you a run down of the common operations first, and the list-specific operations second.

### **Common sequence operations**

* len(sequence) Returns the length of the sequence
* for element in sequence Iterates over each element in the sequence
* if element in sequence Checks whether the element is part of the sequence
* sequence[i] Accesses the element at index i of the sequence, starting at zero
* sequence[i:j] Accesses a slice starting at index i, ending at index j-1. If i is omitted, it's 0 by default. If j is omitted, it's len(sequence) by default.
* for index, element in enumerate(sequence) Iterates over both the indexes and the elements in the sequence at the same time

Check out the [official documentation for sequence operations](https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range).

### **List-specific operations and methods**

* list[i] = x Replaces the element at index i with x
* list.append(x) Inserts x at the end of the list
* list.insert(i, x) Inserts x at index i
* list.pop(i) Returns the element a index i, also removing it from the list. If i is omitted, the last element is returned and removed.
* list.remove(x) Removes the first occurrence of x in the list
* list.sort() Sorts the items in the list
* list.reverse() Reverses the order of items of the list
* list.clear() Removes all the items of the list
* list.copy() Creates a copy of the list
* list.extend(other\_list) Appends all the elements of other\_list at the end of list

Most of these methods come from the fact that lists are mutable sequences. For more info, see the [official documentation for mutable sequences](https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types) and the [list specific documentation](https://docs.python.org/3/library/stdtypes.html#lists).

### **List comprehension**

* [expression for variable in sequence] Creates a new list based on the given sequence. Each element is the result of the given expression.
* [expression for variable in sequence if condition] Creates a new list based on the given sequence. Each element is the result of the given expression; elements only get added if the condition is true.
* The odd\_numbers function returns a list of odd numbers between 1 and n, inclusively. Fill in the blanks in the function, using list comprehension. Hint: remember that list and range counters start at 0 and end at the limit minus 1.
* def odd\_numbers(n):
* return [x for x in range(n+1) if x%2==1]
* print(odd\_numbers(5))  # Should print [1, 3, 5]
* print(odd\_numbers(10)) # Should print [1, 3, 5, 7, 9]
* print(odd\_numbers(11)) # Should print [1, 3, 5, 7, 9, 11]
* print(odd\_numbers(1))  # Should print [1]
* print(odd\_numbers(-1)) # Should print []
* RunReset
* Here is your output:
* [1, 3, 5]
* [1, 3, 5, 7, 9]
* [1, 3, 5, 7, 9, 11]
* [1]
* []
* Excellent! You're using the power of list comprehension to
* do a lot of work in just one line!