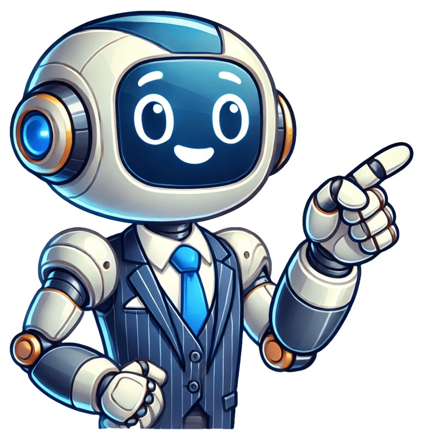
I'm not a bot



/en/computer-science/programming-languages/content/ Sequences, selections, and loops Behind all of the software we use on a daily basis, there's a code being run with all sorts of terms and symbols. Surprisingly, it can often be broken down into three simple programming structures called sequences, selections, and loops. These come together to form the most basic instructions and algorithms for all types of software. Watch the video below to learn more about sequences, selections, and loops. A sequence is a series of actions that is completed in a specific order. Action 1 is performed, then Action 2, then Action 3, etc., until all of the actions in the sequence have been carried out. A sequence we do every day is a morning routine. You might wake up, drink some water, take a shower, eat breakfast, and so on. Everyone's routine is different, but they're all made up of a sequence of various actions. Selections are a bit different. Instead of following a specific order of events, they ask a question in order to figure out which path to take next. Let's say you go to brush your teeth, and you find that you're out of toothpaste. You'd then ask, "Do I have any more toothpaste?" If the answer is no, then you would add it to your shopping list. But if the answer is yes, you would just use the toothpaste. This is really all a selection is doing: answering a question based on what it finds. The third programming structure is a loop. Like selections, loops ask questions. However, the difference is that they ask the same question over and over and over again, until a certain task is complete. For example, take the act of hammering a nail. Even though you may not realize it, you're constantly asking yourself, "Is the nail all the way in?" When the answer is no, you hammer the nail again. You continue to repeat this question until the answer is yes, and then you stop. Loops allow programmers to efficiently code repetitive tasks instead of having to write the same actions over and over again. These three programming structures may seem pretty simple on their own, but when combined they can create some pretty complex software. /en/computer-science/should-i-learn-to-code/content/ Number sequence calculator is amongst the very common mathematics calculators available today. A sequence, basically, refers to some ordered list containing different objects. And, when it comes to number sequence, it contains numbers in the ordered list that follows a certain pattern. The elements of the sequence are called as terms while the length of the sequence is exactly how many terms are there in it. It can even be infinite as well. In any number sequence, the order in which the sequence goes on is very important. And, there can be several terms that can repeat in the sequence on a given pattern. Our number sequence calculator gives you access to three most commonly used sequences namely known as arithmetic, geometric and the Fibonacci sequences. All these sequences can be implemented in different mathematical disciplines because of the convergence properties that they have. In case of a convergent series, the sequence converges to a certain limit and if it doesn't that it's a divergent series. Arithmetic Sequence Calculator S No Beginning Balance Interest Principal Ending Balance Arithmetic SequenceIn arithmetic sequences, the difference between the successive terms remains the same. And if you know the first number in the sequence and the common difference, then our number sequence calculator will give you the value of any given number in the sequence as well as the sum of the sequence until that particular number. Geometric SequenceIn geometric sequences, every next number in the series after first is multiplication of previous number in the series with a non-zero, fixed number. In our number sequence calculator for geometric sequence, you have to provide the first number of the sequence, the common multiplier and the position whose value you want to find out in the series. As a result, you will not just get the value of the given number in the series as well as the sum of series up to that point. Fibonacci SequenceThe first two sequence that we have mentioned above appear to be the very simple ones. However, things get a bit complicated with the Fibonacci sequence. The sequence starts with two numbers after which every next number in the sequence is obtained by adding two numbers it is preceded by. Any Fibonacci sequence starts with either 0, 1 or 1,1 based on the starting point you choose. So, if you use our number sequence calculator for this type of a sequence, all you have to enter is the position whose value you want to find out and the result will show you the sequence itself, the number at the position you've entered and sum of all the numbers in the series until that position. So, use our number sequence calculator to find out the desired information about a given sequence as mentioned above. Summary: in this tutorial, you'll learn about the Python sequences and their basic operations.A sequence is a positionally ordered collection of items. And you can refer to any item in the sequence by using its index number e.g., s[0] and s[1].In Python, the sequence index starts at 0, not 1. So the first element is s[0] and the second element is s[1]. If the sequence s has n items, the last item is s[n-1].Python has the following built-in sequence types: lists, bytearrays, strings, tuples, range, and bytes. Python classifies sequence types as mutable and immutable.The mutable sequence types are lists and bytearrays while the immutable sequence types are strings, tuples, range, and bytes.A sequence can be homogeneous or heterogeneous. In a homogeneous sequence, all elements have the same type. For example, strings are homogeneous sequences where each element is of the same type.Lists, however, are heterogeneous sequences where you can store elements of different types including integer, strings, objects, etc.In general, homogeneous sequence types are more efficient than heterogeneous in terms of storage and operations.An iterable is a collection of objects where you can get each element one by one. Therefore, any sequence is iterable. For example, a list is iterable.However, an iterable may not be a sequence type. For example, a set is iterable but it's not a sequence.Generally speaking, iterables are more general than sequence types.The following explains some standard sequence methods:To get the number of elements of a sequence, you use the built-in len function:len(seq)The following example uses the len function to get the number of items in the cities list:cities = ['San Francisco', 'New York', 'Washington DC'] print(len(cities)) Code language: PHP (php)Try itOutput:3To check if an item exists in a sequence, you use the in operator:element in seqThe following example uses the in operator to check if the 'New York' is in the cities list:cities = ['San Francisco', 'New York', 'Washington DC'] print('New York' in cities)Code language: PHP (php)Try itOutput:TrueCode language: PHP (php)To negate the in operator, you use the not operator. The following example checks if 'New York' is not in the cities list:cities = ['San Francisco', 'New York', 'Washington DC'] print('New York' not in cities)Code language: PHP (php)Try itOutput:FalseCode language: PHP (php)The seq.index(e) returns the index of the first occurrence of the item e in the sequence seq:seq.index(e)Code language: CSS (css)For example:numbers = [1, 4, 5, 3, 5, 7, 8, 5] print(numbers.index(5)) Code language: PHP (php)Try itOutput:2The index of the first occurrence of number 5 in the numbers list is 2. If the number is not in the sequence, you'll get an error:numbers = [1, 4, 5, 3, 5, 7, 8, 5] print(numbers.index(10))Code language: PHP (php)Error:ValueError: 10 is not in listCode language: PHP (php)Try itTo find the index of the first occurrence of an item at or after a specific index, you use the following form of the index method:seq.index(e, i)Code language: CSS (css)The following example returns the index of the first occurrence of the number 5 after the third index:numbers = [1, 4, 5, 3, 5, 7, 8, 5] print(numbers.index(5, 3))Code language: PHP (php)Try itOutput:4The following form of the index method allows you to find the index of the first occurrence of an item at or after the index i and before index j:seq.index(e, i, j)Code language: CSS (css)For example:numbers = [1, 4, 5, 3, 5, 7, 8, 5] print(numbers.index(5, 3, 5))Code language: PHP (php)Try itOutput:4To get the slice from the index i to (but not including) j, you use the following syntax:seq[i:j]Code language: CSS (css)For example:numbers = [1, 4, 5, 3, 5, 7, 8, 5] print(numbers[2:6])Code language: PHP (php)Try itOutput:[5, 3, 5, 7]Code language: JSON / JSON with Comments (json)When you slice a sequence, it's easier to imagine that the sequence indexes locate between two items like this:The extended slice allows you to get a slice from i to (but not including j) in steps of k:seq[i:j:k]Code language: CSS (css)For example:numbers = [1, 4, 5, 3, 5, 7, 8, 5] print(numbers[2:6:2])Code language: Python (python)Try itOutput:[5, 5]Code language: JSON / JSON with Comments (json)If the ordering between items in a sequence is specified, you can use the built-in min and max functions to find the minimum and maximum items:numbers = [1, 4, 5, 3, 5, 7, 8, 5] print(min(numbers)) print(max(numbers)) Code language: PHP (php)Try itOutput:1 8To concatenate two sequences into a single sequence, you use the + operator:s3 = s1 + s2The following example concatenates two sequences of strings:east = ['New York', 'New Jersey'] west = ['San Diego', 'San Francisco'] cities = east + west print(cities)Code language: PHP (php)Try itOutput:['New York', 'New Jersey', 'San Diego', 'San Francisco']Code language: JSON / JSON with Comments (json)It's quite safe to concatenate immutable sequences. The following example appends one element to the west list. And it doesn't affect the cities sequence:east = ['New York', 'New Jersey'] west = ['San Diego', 'San Francisco'] cities = east + west west.append('Sacramento') print(west) print(cities)Code language: PHP (php)Try itOutput:['San Diego', 'San Francisco', 'Sacramento'] ['New York', 'New Jersey', 'San Diego', 'San Francisco']Code language: JSON / JSON with Comments (json)However, you should be aware of concatenations of mutable sequences. The following example shows how to concatenate a list to itself:city = [['San Francisco', 900_000]] cities = city + city print(cities)Code language: PHP (php)Output:[['San Francisco', 1000000], ['San Francisco', 1000000]]Code language: JSON / JSON with Comments (json)Since a list is mutable, the memory addresses of the first and second elements from the cities list are the same:print(id(cities[0]) == id(cities[1])) Code language: PHP (php)In addition, when you change the value from the original list, the combined list also changes:city[0][1] = 1_000_000 print(cities)Code language: PHP (php)Putting it all together:city = [['San Francisco', 900_000]] cities = city + city print(cities) print(id(cities[0]) == id(cities[1])) city[0][1] = 1_000_000 print(cities)Code language: PHP (php)Try itOutput:[['San Francisco', 900000], ['San Francisco', 900000]] True [['San Francisco', 1000000], ['San Francisco', 1000000]]Code language: CSS (css)To repeat a sequence a number of times, you use the multiplication operator (*). The following example repeats the string Python three times:s = 'ha' print(s*3) Code language: PHP (php)Try itOutput:hahahaPython sequences are positionally ordered collections of items.Was this tutorial helpful ? A phrase you'll often hear is that everything in Python is an object, and every object has a type. This points to the importance of data types in Python. However, often what an object can do is more important than what it is. So, it's useful to discuss categories of data types and one of the main categories is Python's sequence. In this tutorial, you'll learn about: Basic characteristics of a sequence Operations that are common to most sequences Special methods associated with sequences Abstract base classes Sequence and MutableSequence User-defined mutable and immutable sequences and how to create them This tutorial assumes that you're familiar with Python's built-in data types and with the basics of object-oriented programming. Take the Quiz: Test your knowledge with our interactive "Python Sequences: A Comprehensive Guide" quiz. You'll receive a score upon completion to help you track your learning progress: Interactive Quiz Python Sequences: A Comprehensive Guide In this quiz, you'll test your understanding of sequences in Python. You'll revisit the basic characteristics of a sequence, operations common to most sequences, special methods associated with sequences, and how to create user-defined mutable and immutable sequences. It's likely you used a Python sequence the last time you wrote Python code, even if you don't know it. The term sequence doesn't refer to a specific data type but to a category of data types that share common characteristics. A sequence is a data structure that contains items arranged in order, and you can access each item using an integer index that represents its position in the sequence. You can always find the length of a sequence. Here are some examples of sequences from Python's basic built-in data types: Lists, tuples, and strings are among Python's most basic data types. Even though they're different types with distinct characteristics, they have some common traits. You can summarize the characteristics that define a Python sequence as follows: A sequence is an iterable, which means you can iterate through it. A sequence has a length, which means you can pass it to len() to get its number of elements. An element of a sequence can be accessed based on its position in the sequence using an integer index. You can use the square bracket notation to index a sequence. There are other built-in data types in Python that also have all of these characteristics. One of these is the range object: You can iterate through a range object, which makes it iterable. You can also find its length using len() and fetch items through indexing. Therefore, a range object is also a sequence. You can also verify that bytes and bytearray objects, two of Python's built-in data structures, are also sequences. Both are sequences of integers. A bytes sequence is immutable, while a bytearray is mutable. In Python, the key characteristics of a data type are determined using special methods, which are defined in the class definitions. The special methods associated with the properties of sequences are the following: .__iter__(): This special method makes an object iterable using Python's preferred iteration protocol. However, it's possible for a class without an .__iter__() special method to create iterable objects if the class has a .__getitem__() special method that supports iteration. Most sequences have an .__iter__() special method, but it's possible to have a sequence without this method. .__len__(): This special method defines the length of an object, which is normally the number of elements contained within it. The len() built-in function calls an object's .__len__() special method. Every sequence has this special method. .__getitem__(): This special method enables you to access an item from a sequence. The square brackets notation can be used to fetch an item. The expression countries[0] is equivalent to countries.__getitem__(0). For sequences, .__getitem__() should accept integer arguments starting from zero. Every sequence has this special method. This method can also ensure an object is iterable if the .__iter__() special method is missing. Therefore, all sequences have a .__len__() and a .__getitem__() special method and most also have .__iter__(). However, it's not sufficient for an object to have these special methods to be a sequence. For example, many mappings also have these three methods but mappings aren't sequences. A dictionary is an example of a mapping. You can find the length of a dictionary and iterate through its keys using a for loop or other iteration techniques. You can also fetch an item from a dictionary using the square brackets notation. This characteristic is defined by .__getitem__(). However, .__getitem__() needs arguments that are dictionary keys and returns their matching values. You can't index a dictionary using integers that refer to an item's position in the dictionary. Therefore, dictionaries are not sequences. All sequences support indexing using an integer that represents the item's position within the sequence. This requirement is part of the definition of a Python sequence. Most sequences also support slicing, which is often closely associated with indexing. You can slice most sequences to access a subset of the elements. All of the sequences you've encountered so far can be sliced: In all four examples, you extract the items from index 1 up to but excluding index 4. The result is a data structure of the same type as the original one containing the subset of elements. Slicing also depends on the .__getitem__() special method in these data types and other sequences. Typically, .__getitem__() can accept either an integer or a slice object. The special method's behavior depends on whether it's passed an integer or a slice. However, it's possible to have sequences that don't support slicing. The deque data type in Python's collections module is an example of this. First, you can confirm that a deque is a sequence: You can iterate through a deque, get its length, and index it. Therefore, it's a sequence but you can't slice a deque: The TypeError shows that a deque can only be indexed using an integer, and a slice can't be used in the square brackets. Deques are optimized to provide efficient access at either the beginning or the end of the data structure. Therefore, slicing would be inefficient and generally not the intended use for deques. Although most sequences support slicing, you shouldn't assume that all of them do. Most sequences can be added to another sequence of the same type. For example, you can combine two lists, or you can add a tuple to another tuple: The output is a sequence of the same type as the original ones. However, in general, you can't add sequences of different types: You get an error when you try to add a list and a tuple. Many sequences can be concatenated in this way, but not all of them. Here's an example of a sequence that can't be concatenated: You learned earlier that a range object is a sequence. However, it requires items that follow specific patterns that can be represented by a start, stop, and step value. For this reason, a range object can only represent a series of numbers with regular increments between them. In the example above, the first range object represents the numbers from one to nine. The second range object represents the numbers from ten to nineteen but in steps of two. Therefore, you can't represent the concatenation of these two series using a range object, which must be defined by its start, stop, and step values. Another difference in behavior between sequences is highlighted when using the augmented assignment operators, such as +=, on mutable and immutable data types: The extended list is the same object as the original numbers_list. The value returned by the built-in id() function is the same before and after the augmented assignment operation. However, the behavior is different when using tuples: The augmented addition assignment applied to a list, which is a mutable sequence, extends the same object with the new values. However, tuples are immutable, which leads the += operator to create a new object. Built-in sequences support value comparisons such as equality and greater than or less than comparisons. However, only sequences of the same type can be compared: When two sequences of the same type are compared to check if one is greater than or less than the other, the first non-equal value determines the outcome: Both lists have the same first two values. However, the third item in numbers is larger than the third item in more_numbers. Therefore, numbers is greater than more_numbers even though the remaining integers in more_numbers are larger. If the items in a sequence are equal to items with matching indices in another sequence, but one sequence has more items, the sequence with more items is considered greater: However, in the second example, numbers[1] is 0, which is smaller than more_numbers[1]. As a result, more_numbers is greater than numbers. You've learned about features that define sequences, including the special methods they have in common. You can also use this knowledge to create user-defined classes that are sequences. In this section, you'll define a class called ShapePoints, which contains a number of points that define the vertices of a shape. You can create a file named shape.py: The class's .__init__() includes a points parameter. You pass a sequence of coordinate pairs when you create a ShapePoints object, such as a list of tuples. You then cast the input sequence as a new list object to avoid some mutability issues and assign it to the data attribute .points. Now you can create instances of this new class and explore its features through a REPL session: You import the class ShapePoints and create an instance with three points. The shape represents a triangle, and the data attribute .points contains the three tuples with the points' coordinates. You can make a few more additions to the basic setup for this class before you start exploring its sequence features. This class is used for closed shapes, which means the last point should be identical to the first. You can ensure this is the case by adding an extra point that's equal to the first if it doesn't already exist. You also add a docstring with a basic description of the class: A ShapePoints object is always closed. Therefore, you add the first point to the end of the list .points if the first and last points aren't already equal. This step ensures the first and last vertices are the same. As you explore this class in this tutorial, you'll make the necessary changes to keep the shape closed even when you modify the points in the shape. Next, you add a .__repr__() special method to define a string representation for the class: The .__repr__() special method ensures a meaningful output in all situations when you display the object. It's always a good idea to include this special method in user-defined classes. Note: You'll need to start a new REPL session each time you make changes to the class definition in shapes.py. You can't write the import statement again in the same REPL session, as this won't import the updated class. It's also possible to use importlib from the standard library to reload a module, but it's easier to start a new REPL. In a new REPL session, you can create the ShapePoints object again and display the object directly instead of its .points attribute: The output shows the string representation for the ShapePoints object, which includes four points since it's a closed shape and the first point is repeated at the end. Say you would like a ShapePoints object to be a sequence. However, it doesn't meet any of the three requirements at the moment. You can try to iterate through triangle: When you try to iterate through triangle, Python raises an exception since the object is not iterable. You can try to find the length of the object: Passing a ShapePoints object to len() also doesn't work and leads to another TypeError. The error states that the object has no length. Finally, you can attempt to index the object to retrieve one of its elements: Attempting to index triangle completes the trio of TypeErrors. The error now states that a ShapePoints object is not subscriptable, which means you can't use the square brackets notation to access values within it. You can add the .__getitem__() special method to the class in shape.py to make it subscriptable. However, if you want a ShapePoints object to be a sequence, this special method needs to accept integers and fetch items based on their position: You rely on the fact that the .points data attribute is a list, which also makes it a sequence. Now you can check whether indexing works in a new REPL session: A ShapePoints object is now indexable since you can use an integer to fetch an item based on its position. Since you're using the sequence features of the list data type in .points, you can also slice a ShapePoints object: This notation returns a slice of the ShapePoints object, which in this case is the slice that includes the first and second elements of triangle. Adding the .__getitem__() special method makes a ShapePoints object indexable. However, it also enables other features that use this method. For example, you can now iterate through the ShapePoints object: This code no longer raises a TypeError, as it did earlier in this tutorial. Instead, it prints the tuples with the shape's points, including the additional final point that's equal to the first point since the shape is closed. Later in this tutorial, you'll update the class to make it iterable using another special method, .__iter__(). Using .__iter__() is the preferred option to make objects iterable in modern Python. You can check whether an element is a member of triangle, and you can also sort the items in the shape: The code outputs True to show that (100, 100) is a member of the ShapePoints object. It also outputs a list with the points sorted following the default sorting rules for tuples. Note that sorted() returns a list of tuples and not a ShapePoints object. Adding .__getitem__() adds several features to the ShapePoints class. However, there are other special methods dedicated to these features, which you'll explore later. You still can't get the length of a ShapePoints object. In the next section, you'll add another special method to define the length of an object. A sized object is one that has a defined length, which you can access using the built-in len() function. To make a ShapePoints object sized, you can add the .__len__() special method to the class definition. This method must return an integer. You'll need to decide whether you want the length of the ShapePoints object to include all the values in .points or whether you want to exclude the final pair of coordinates from the count so that the length represents the number of vertices. This version defines the length as the number of points in the shape: Since .points is a list and has a length, you subtract one from the length of the list to account for the repeated point at the end of .points. You can now call len(triangle) and get the number of points in the shape. Remember, you'll need to start a new REPL session to explore the updated class: The .__len__() special method also provides the object with a definition of truthiness. You can convert any object to a True or False value using the built-in bool(). By default, an object is truthy, which means it's converted to True when passed to bool(). However, a sized object is truthy if it has a non-zero length and falsy if it's empty. You can convert triangle into a Boolean using bool(): To confirm that the object follows the truthiness rules for sequences, you'll need to create an empty ShapePoints object: This step highlights a bug in your code that occurs when there are no points in the shape. Since the shape is closed, you duplicate the first point to place it at the end of the sequence. However, points[0] raises an error when the list is empty. You can update the if statement in the class's .__init__() method to account for this case: In the expression in the if statement, if points is empty, it evaluates as falsy. The and expression is evaluated lazily, which means that if points is falsy, the rest of the and expression is ignored and won't raise an exception. You can now create an empty ShapePoints object in a new REPL to confirm that an empty object is falsy: However, this raises yet another exception. The .__len__() special method subtracts one from the length of .points, but the value returned by .__len__() can't be negative since a negative length is invalid. You can update .__len__() to account for this scenario: If .points is not empty, you subtract the last value and return the result. If there are no values in .points, you return zero. A new REPL session confirms that an empty ShapePoints object is falsy: You can comment out the .__len__() definition in shape.py and run the same code in a new REPL. Without the .__len__() special method, the object is always truthy. The ShapePoints object is already a sequence since it's iterable, indexable, and has a length. However, it's preferable to use the .__iter__() iteration protocol to make an object iterable since this offers more compatibility across different types of iteration and can provide more efficient iteration. You can define this special method and rely on the list in .points to provide the iteration: The .__iter__() special method must return an iterator, which is used in iteration protocols such as the for loop. You create an iterator from the list .points using the built-in iter() function. Although adding .__iter__() is not necessary to ensure a ShapePoints object is a sequence, most iterables implement this special method. You can visualize the shape with the help of the turtle module, which is the simplest way to display graphics using Python's standard library. Now you can create a new script to run this code: Copied! The output shows the shape defined by the points in the ShapePoints object: You'll continue to work with the ShapePoints class in the following section as you learn about another way to create user-defined sequences. You've learned about the minimum requirements for an object to be a sequence. A sequence must be iterable, have a length, and be indexable. Many sequences can also be sliced but this is not a feature that's universal to all sequences. There are other features that are common to many sequences, even though they're not required. The abstract base class collections.abc.Sequence provides a template that defines the interface for most sequences. This abstract base class goes further than the minimum requirements. You can use this abstract base class to confirm whether a data type is a sequence: You confirm that lists and tuples are sequences, but sets and dictionaries aren't. However, the Sequence abstract base class sets a higher bar for defining sequences than the basic definition you learned earlier. You can try to check whether ShapePoints is a sequence: Earlier in this tutorial, you added special methods to ShapePoints to ensure it meets the requirements for a sequence. However, it doesn't meet the higher standards set by the Sequence abstract base class. You'll explore these additional features shortly. There are two apparent definitions of a sequence, which may lead to confusion. In practice, this is rarely an issue, and most sequences meet the higher standards set by the abstract base class Sequence. You can redefine the ShapePoints class in a new file called shape_abc.py. In this version, the class inherits from the Sequence abstract base class. Start by defining the class's .__init__() special method as in the earlier version: Copied! The new class inherits from the abstract base class Sequence. Try to create an instance of this class in the REPL. Note that you're now importing from the new module shape_abc.py: When you try to create an instance of ShapePoints, you get an error. You can't have a sequence without the .__getitem__() and .__len__() methods. You need to define these in a way that suits your class before you can proceed. You can use the same definitions you used in the earlier version, and you can also add .__repr__() and .__iter__(): Copied! The ShapePoints object now meets the stricter standards set by the Sequence abstract base class. You can confirm this in a new REPL session: What does this mean? What extra features does ShapePoints have now that it's a subclass of the collections.abc.Sequence abstract base class? You'll explore the answers to these questions in the following sections. Objects of different data types have different methods. However, some data types also share methods with similar names. For example, lists, strings, and tuples all have a .index() method to find the position of an item and a .count() method to find how often an item occurs in the data structure. In fact, these two methods are the only methods tuples have that aren't special methods. Most sequences have access to .index() and .count() methods. These methods are part of the interface provided by collections.abc.Sequence. You can confirm this by calling these methods on the ShapePoints instance you created in the previous section: The object triangle has access to these methods even though you don't define them when you create the ShapePoints class. The first output shows the index of the item (100, 100), which is 0. The .index() method returns the index of its argument's first occurrence if there's more than one. This method relies on .__getitem__() to fetch each item until the required value is found, or it raises a ValueError if the value is not present. The first element is repeated at the end of the sequence since shapes are closed. Therefore, .count() returns 2 as there are two occurrences of (100, 100). This method relies on .__iter__() to iterate through the whole sequence and count the occurrences of the value passed as an argument. If .__iter__() is not defined, then .__getitem__() is used instead. However, you don't want to count the first point in the shape twice, so you can override the .count() method: You create your own .count() method rather than using the one inherited from the Sequence abstract base class. You exclude the last point in the shape by counting the number of occurrences of the required value in .points[:-1]. The slice includes all the points except the one with index -1, which is the last element. You can refresh the REPL to make sure this works: The .index() and .count() methods are included when you create a sequence by inheriting from the abstract base class. However, you can also override the default methods by defining your own versions. Sequences usually also have two more characteristics: They are containers. They are reversible. In this section of the tutorial, you'll explore these two data types characteristics and how to ensure your user-defined sequences are also containers and reversible. Many data structures are also containers, which means that Python can determine whether an element is a member of the data structure. A common way to find out whether a data structure contains an element is to use the in keyword: However, this is not sufficient to confirm that a data structure is a container. For example, if you use the in keyword on an iterator, you may not get the results you expect: You create an iterator from the list countries. Python returns True the first time you check whether Canada is one of the countries included in the data structure. But an iterator generates items when they're required and doesn't store them. So, the second time you check for membership, Python returns False. That being the case, iterators are not containers. You can control the definition of membership of an element using the .__contains__() special method. This is one of the methods that's included in the Sequence abstract base class. You can confirm that the class ShapePoints you created, which inherits from Sequence, has this special method: You call the special method .__contains__() directly on triangle to confirm this special method exists even though you didn't define it in the class. It's inherited from the Sequence abstract base class. Keep in mind that it's best to avoid calling the special method directly. Special methods are intended to be called behind the scenes. For example, the .__contains__() special method is called when you use the in operator if the special method is available. If you have specific requirements for what membership means in a user-defined class, you can override this method: Copied! You define the special method in the class definition. In this example, you add a call to print() to highlight when the program calls this method. When you repeat the statements from the previous REPL session, you'll also see this text displayed: The extra sentence is printed when you call .__contains__() directly and when you use the in operator, which confirms that both expressions are equivalent. Another special method that's included in the abstract base class Sequence is .__reversed__(). This method defines how a sequence can be reversed, and it's called by functions such as the built-in reversed(). Note: The .__contains__() and .__reversed__() special methods aren't necessary for an object to be a container and reversible. The .__getitem__() special method does a lot of heavy lifting, and Python will fall back onto this method to determine whether an item is a member of a data structure or to reverse the sequence. You can confirm this with the earlier version of ShapePoints you created, which doesn't inherit from Sequence and, therefore, doesn't include these special methods. However, these methods make the intention clearer and allow you to customize your class's behavior. All sequences are containers and are reversible, and they include the .__contains__() and .__reversed__() special methods if they follow the collections.abc.Sequence requirements. You defined a class called ShapePoints, which inherits from the abstract base class collections.abc.Sequence. Next, you try to modify one of the points in an instance of ShapePoints: The expression triangle[1] returns the tuple representing the second point in the shape. This behavior is defined by the .__getitem__() special method. However, when you try to assign a new tuple to the second position of your sequence, the program raises an exception. The sequence you create is immutable. Immutable built-in sequences include tuples and strings. Therefore, the sequence you create is similar to these built-in sequences since you can't make changes to an object once it's created. None of the methods and special methods available to the class make changes to the data stored within the sequence. All methods, whether you write them or they're inherited from Sequence, return values without modifying the state of the object. Note: Special methods such as .__init__() and .__new__() do make changes to the object. But these are only called when you create a new instance since their purpose is to create a new object and initialize it. Still, it's possible to have mutable sequences. The built-in list is the most common example of a mutable sequence. You'll learn how to create a user-defined class for a mutable sequence in the following section. You can make your custom sequence mutable by inheriting from a different abstract base class called collections.abc.MutableSequence : Copied! You import MutableSequence instead of Sequence in shape_abc.py. The class ShapePoints now inherits from MutableSequence. However, you'll get an error when you try to create an instance of this class in a new REPL session: You can't have a mutable sequence without defining three new methods: .__delitem__(): This special method defines what should happen when an item is deleted from the sequence, such as when you use the del keyword. .__setitem__(): This special method defines the object's behavior when you assign a value to a position in the sequence, such as when you reassign a new value to replace an existing one. .insert(): This method defines the behavior when you insert a new value in an existing sequence. The data in a ShapePoints object is stored in a list within the data attribute .points. A list is a sequence, so you can use the list's mutable properties: Copied! You define deleting an item from the sequence as the same as deleting the corresponding item from the list .points. The operations to set and insert an item also use the equivalent operations on the list .points. Adding these methods is sufficient to make the sequence mutable. You can also improve the logic in these methods to account for the special requirements for a closed shape, where the first and last elements are the same. You can account for this requirement in the definitions of the new methods. To start, update .__delitem__(): Copied! Whenever you delete the first or last elements, which represent the same point, you need to delete the first element and replace the last element so it's equal to the sequence's new first element. Ideally, you should also ensure that the sequence isn't empty before deleting an element, but you'll change how you handle such sequences shortly. You implement this functionality in a few steps: In line 7, you check whether the index that you pass to .__delitem__() is either 0, -1, or the last index of self.points. If it is, then you're dealing with the point that's duplicated to represent a closed shape. If the expression in line 7 evaluates to True, then you delete the first point of your custom sequence in line 8. This removes the duplicate point at the beginning of your sequence. Next, in line 8, you remove the duplicate point from the end of the sequence by replacing it with the value of the new first point. That way, you've successfully removed both instances of the duplicate point. If you want to delete any other point of the sequence, then you delete it without any additional actions in the else block. You can confirm this works in a new REPL session. You add a few more points to the shape to make it easier to test the changes you made: You start with a shape with five points. Since shapes are closed, the object actually has six elements as the first point is duplicated at the end of the sequence. Next, you delete an element that isn't at either end of the sequence with the statement del polygon[1]. Only the second point is deleted. When you remove the first or last element, both ends are updated to reflect the new shape. Note that in the last step, the shape only has two points, since the last element is always equal to the first. This shape represents a line, which is not a closed shape. A ShapePoints object with only one point represents that point, which is also not a closed shape. If you prefer, you could define ShapePoints to be valid only for three or more points to ensure you have a closed shape: You add a class attribute .MIN_POINTS to define the minimum number of points an object can have. Then you update both .__init__() and .__delitem__() to raise an error when there's an invalid number of points. When you initialize the object, you confirm that the iterable you pass to create the instance contains at least three points since .MIN_POINTS is 3 in this example. When you call .__delitem__(), you check that there are at least four points before deleting one. You can confirm these changes in a new REPL session: You get a ValueError when you try to create a line with only two points. You also get the same exception when you try to delete a point from a triangle since a ShapePoints object can't have fewer than three points. Note that the adjustment you made earlier to .__len__() to ensure it works for empty sequences isn't needed now since a ShapePoints object can no longer be empty. You also need to account for the special requirements of a closed shape when you modify a point. Therefore, you need to update .__setitem__() similarly to how you modified .__delitem__(): Copied! You update both the first and last elements of the sequence when the user sets either the first or the last element. For any other element, you only set the value required. The following examples in a new REPL session confirm that .__setitem__() works the way you expect: Only the second element is updated when you assign a new value to polygon[1]. However, you modify both the first and last values when you update polygon[0]. You also modify both the first and last values when you change the last element in the ShapePoints object. Finally, you update the .insert() method: Copied! Any value inserted in the first or last position updates both values. If the value updated is not the first or last element, you can use the standard insertion at the required position. You can confirm that the changes to .insert() achieve the outcome required. In this example, you start with a shape with fewer points in a new REPL session: Your code treats insertion at the first and last positions differently to ensure the new shape remains closed. When a class inherits from collections.abc.MutableSequence, it also inherits several other methods. You'll recognize some or all of these methods from your experience with using lists: .append(): Add a new item to the end of the sequence. .clear(): Remove all items from the sequence. .reverse(): Reverse the items of the sequence in place, changing the existing object rather than returning a new one. .extend(): Add several additional items to the end of the sequence by passing another sequence as an argument. .pop(): Remove an item based on its index. This method returns the item that's removed from the sequence. .remove(): Remove an item based on its value. This method removes the first occurrence of the value in the sequence. .__iadd__(): This special method defines the behavior for the augmented addition operator +=, which for mutable sequences becomes an in-place operator. All of these methods are included by default when you inherit from MutableSequence. However, you may need to override their behavior if your sequence has certain non-standard requirements, such as the ShapePoints sequence. You can test these methods on the current version of ShapePoints: You start with a shape with two points. You call .append() and .extend() to add points to the shape. Next, you remove a point, use the augmented addition operator += to extend the sequence again, and finally reverse the sequence. The code doesn't raise any exceptions since all the methods exist. However, not all these methods behave the way you might expect them to. Currently, .append() adds a point at the beginning and the end of the sequence. For this implementation of ShapePoints, that's not what you want .append() to do. The problem occurs because you defined the class with special behavior when new points are added at the end. So, when you update the last element using the default .append() method inherited from the abstract base class, the ShapePoints class also updates the first element. You can see similar odd behavior with .extend() and the += operator since these operations rely on .append(). The new points are added to the beginning and the end of the shape. When the default methods you inherit from the abstract base class aren't suitable, you can override them. However, it may not be necessary to define all the methods. With mutable sequences, .append() is often a method you may wish to update first when the defaults aren't sufficient: Copied! You redefine .append() to add a new point before the final element since the final element mirrors the first one. Therefore, you call .append() on the list stored in the data attribute .points and add the first element of the sequence. Now, you can update the element with index -2, which is no longer the last element of the sequence, but it's the last point in the shape. You can verify what happens to the same operations you tried earlier in a new REPL session: When you append a new point to a ShapePoints sequence, the additional point is added to the shape, but the final element remains matched to the first element. This correct behavior now applies to .extend() and the += operator. So, overriding the .append() method also deals with these operations since they call .append() behind the scenes. All operations now work as intended. The abstract base classes Sequence and MutableSequence provide you with the methods you need to create custom sequences. When you define a class that inherits from Sequence, you need to define at least two special methods: .__getitem__() .__len__() If you need a mutable sequence and your class inherits from MutableSequence, you'll also need at least three more methods in addition to the two special methods listed above: .__setitem__() .__delitem__() .insert() All sequences will also have .index() and .count() available out of the box, and mutable sequences have more methods you can use or override if you require special behavior. Here's the final version of the ShapePoints class definition you wrote in this tutorial: Copied! Different data types often share common traits, and it's useful to categorize them based on their shared features. Sequences are data types that contain ordered items which can be accessed using an integer index. In addition to the basic requirements that make an object a sequence, there are other features that are present in many sequences. In this tutorial, you learned about: Basic characteristics of a sequence Operations that are common to most sequences Special methods associated with sequences Abstract base classes Sequence and MutableSequence User-defined mutable and immutable sequences and how to create them You're now better equipped to use all the data types that fall under the sequences category, and you know how to deal with functions requiring arguments that are sequences. You're also ready to craft the ideal custom classes whenever you need to create your own mutable or immutable sequences. Take the Quiz: Test your knowledge with our interactive "Python Sequences: A Comprehensive Guide" quiz. You'll receive a score upon completion to help you track your learning progress: Interactive Quiz Python Sequences: A Comprehensive Guide In this quiz, you'll test your understanding of sequences in Python. You'll revisit the basic characteristics of a sequence, operations common to most sequences, special methods associated with sequences, and how to create user-defined mutable and immutable sequences. I have an admission to make.I've used the terms iterable and sequence interchangeably in the past for longer than I wish to admit. You can get away with this in the early days of learning to code in Python. They're quite similar……until you dig deeper beneath the surface, which is what we'll do in this article.This is the second of seven articles in this series. You can read the first one about iterables if you missed it. Here's the overview of the series:IterableSequence (this article)MappingContainerCollectionIteratorGeneratorThere's a reason why it's easy to confuse iterables and sequences. All sequences are iterables. We'll talk more about this later. And you're likely to see common data structures such as lists and tuples referred to as sequences sometimes and as iterables other times.Let's start with the headline difference between the two terms:A Python sequence is an iterable that you can index using an integer.This means you can use an integer inside square brackets to get an item from a sequence, such as some_sequence[0]. You can also use slices within the square brackets.In the first article in the series, we discussed how an iterable is an object that can return its elements one at a time. With a sequence, we're going further. You can fetch an item based on its position in the sequence.Let's look at some examples of sequences:Lists, strings, and tuples are among the most common sequences.There's another requirement for an object to be a sequence. It needs to have a length:You may think this is obvious and that every data structure must have a length. However, later in this series, we'll look at iterables that don't have a length.All sequences are iterables. But not all iterables are sequences.Let's take a dictionary, for example. In the first article in the series, we determined that a dictionary is an iterable. Although you could use an integer in the square brackets to fetch an item if that integer is one of the dictionary's keys, you can also use non-integer data types as keys. To put this in another way, you cannot fetch the second item in a dictionary by using my_dictionary[1].Therefore, a dictionary is an iterable but not a sequence.Let's look at some other data types that are not sequences. Let's start with sets:We've created a set and checked that it's iterable. We've checked using two techniques for good measure—using the set in a for loop and passing it to iter(). These are not really different checks since when you use an object in a for loop, it's converted to an iterable using iter().Therefore, sets meet one of the criteria for being a sequence. How about the "length test"?A set has a length. Therefore, it passes the "length test", too. But there's one more test it needs to pass:A set cannot be indexed with an integer. The TypeError tells us that a set is not subscriptable—it cannot be indexed!Let's explore another data type:You create a zip object using zip(). This object fails both the "length" test and the "indexed with an integer" test. Therefore, zip objects are not sequences. However, they are iterables:So, the zip object is another example of an iterable that's not a sequence. We'll look at what category zip objects belong to later in this series.I'll finish this article by bringing everything together to see what makes an object a sequence.In the first article in the series, we discussed how for a class to create iterables, it must have at least one of __iter__() or __getitem__() defined. This also applies to sequences, since a sequence is also an iterable.However, a sequence can be indexed, and it must be indexed with an integer or a slice. The __getitem__() special method makes the instances of a class indexable. Therefore, a class needs to define this method to be a sequence. And the implementation of __getitem__() should ensure that it accepts only integers or slices.Finally, a sequence needs to have a length. You can use the __len__() special method to define the length of an object.Let's put all of this together. The minimum requirement to make an object a sequence is to define the following special methods for the class:__getitem__(), which makes the object indexable. It should only take an integer argument (or a slice)__len__(), which defines the length of the objectThe __getitem__() special method also makes the object iterable. However, defining __iter__() is preferable to make an object iterable. A sequence should ideally also have the __iter__() method defined.As is often the case, there's more to say about this topic. However, I'll return to fill in some blanks once I've covered a few more data structure categories later in this series.But I'll give you a preview of what's coming next:We'll talk about this diagram later in the series. However, looking at the two categories we've already discussed, iterables and sequences, you'll see they're in different parts of the hierarchical structure.The term "sequence" comes from the Latin sequi, which means "to follow".Therefore, each item in a sequence follows another. That's why you need to use integer indices!Next in the series: mappingCode in this article uses Python 3.11Recently published articles on The Stack:I take back what I said last time! I did say I'm still experimenting with format and how to publish on Substack. In the last couple of articles I mentioned that I may not email all articles in series. I've had a couple of discussions with readers and other writers, and I've changed my mind. I will email most articles now, but I'm also revising my planned schedule of publication. See next bullet pointOriginally I planned to publish weekly on Wednesdays plus another article per week on some weeks. I'm now aiming for a five-day cycle. There will roughly be one article every five days. However, as I've mentioned in my introductory post, I will not publish just for the sake of meeting a self-imposed dealine. So there may be times when the frequency of publication will changeIn other news, the first cohort of The Python Coding Programme is underway. It's fun guiding a small group of very keen and eager learners through the fundamentals of Python. Next cohort starts in mid-May!Do get in touch on Notes (or other platforms) so we can continue the conversation from this and other articles. As those of you who've interacted with me on any social media platform know, I enjoy having conversations on these platforms! Welcome to the world of coding, where every line, character, and symbol carries meaning and power. As a beginner, it's essential to understand the fundamental concepts that form the building blocks of coding. One such fundamental concept is sequencing. Sequencing in coding refers to the arrangement and order in which instructions are written and executed. It involves determining the sequence of steps necessary to achieve a desired outcome or perform a task. Just as a symphony follows a carefully crafted sequence of musical notes, coding relies on a coherent sequence of instructions to achieve the desired result. Whether you're learning a programming language, developing software, or building a website, sequencing plays a crucial role in ensuring that the code operates as intended. Without proper sequencing, the code may not execute correctly or produce the desired output. Why is understanding sequencing important in coding? Well, imagine you're following a recipe to bake a cake. If you mix the ingredients in the wrong sequence or skip a step, the cake might not turn out as expected. Similarly, in coding, each line of code plays a specific role, and the order in which these lines are written determines the outcome. Sequencing is not limited to a single instruction or line of code; it extends to the overall structure and flow of the program. It ensures that each action is executed in the correct order, allowing the code to function smoothly and efficiently. In this article, we will delve deeper into the concept of sequencing in coding, explore its importance, understand how it is used, and provide examples that illustrate its practical application. What is sequencing in coding? Sequencing in coding refers to the arrangement and order in which instructions are written and executed. It involves organizing a series of steps or actions in a logical manner to achieve a specific outcome or perform a task. Just like following a recipe, sequencing in coding ensures that each line of code is executed in the correct order. At its core, coding is about giving instructions to a computer. These instructions are written using programming languages such as Python, Java, or HTML. To create a functioning program or application, developers need to carefully determine the sequence of instructions required to achieve the desired result. Sequencing not only determines the order in which the instructions are executed but also influences the flow and logic of the code. It ensures that each action is performed in a specific sequence, allowing the code to perform the intended tasks. Consider a simple example of displaying a greeting message on a website. To achieve this, the code may involve multiple steps such as defining variables, assigning values, and displaying the message. Proper sequencing ensures that each step is executed in the correct order, resulting in the desired output. Sequencing is not only limited to a linear progression of instructions. It also involves branching and looping, allowing code to make decisions and repeat actions according to specific conditions. Without sequencing, code becomes disorganized and ineffective. It may produce errors, unexpected results, or fail to accomplish the desired outcome. Therefore, understanding and implementing proper sequencing techniques is fundamental to successful coding. In the next section, we will explore the importance of sequencing in coding and how it impacts the functionality and reliability of a program. Why is sequencing important in coding? Sequencing is a fundamental concept in coding that plays a crucial role in the functionality and reliability of a program. Here are several key reasons why sequencing is important in coding: 1. Logical Execution: Sequencing ensures that instructions are executed in a logical and orderly manner. By following a specific sequence, the code can perform tasks step by step, ensuring that each action is completed before moving on to the next. 2. Desired Outcome: Proper sequencing helps achieve the desired outcome of a program. It allows developers to outline the necessary steps to accomplish a specific task or solve a problem. Without sequencing, the code may not produce the expected results. 3. Code Readability and Maintainability: Sequencing enhances the readability and maintainability of code. When instructions are organized in a logical sequence, it becomes easier for developers to understand and modify the code as needed. This is particularly important when working on collaborative projects or when troubleshooting and debugging code. 4. Error Detection: By following a sequence, it becomes easier to identify any errors or bugs in the code. When each action is executed in the correct order, it allows developers to pinpoint where issues may occur, making it easier to debug and fix problems. 5. Efficiency: Sequencing in coding ensures the efficient execution of a program. By organizing instructions in a logical order, unnecessary repetitions and redundancies can be avoided, leading to optimized code performance. 6. Scalability: When developing complex programs or applications, sequencing becomes even more critical. As the codebase expands, proper sequencing allows developers to integrate new functionalities seamlessly and maintain a structured and scalable codebase. Overall, sequencing in coding is vital for creating programs that work as intended, are easy to understand and modify, and perform efficiently. By following a well-organized sequence of instructions, developers can ensure the success and reliability of their code. How is sequencing used in coding? Sequencing is a fundamental concept in coding that is used in various ways to create structured and functional programs. Here are some common use cases of sequencing in coding: 1. Instructions Execution: Sequencing is used to determine the order in which instructions are executed within a program. Each line of code is written and organized in a specific sequence to ensure that actions are performed in the desired order. 2. Program Flow: Sequencing helps define the flow and logic of a program. It allows developers to control the order in which different parts of the program are executed, based on specific conditions or user interactions. 3. Control Structures: Sequencing is crucial in implementing control structures like conditionals and loops. These structures allow the code to make decisions and repeat actions, depending on certain conditions. By properly sequencing these structures, developers can control the program's behavior. 4. Procedural Programming: In procedural programming, sequencing is used to organize code into functions or procedures. By defining the sequence of steps within each function and the order in which functions are called, developers can create modular and reusable code. 5. Event Handling: When dealing with user interactions or event-driven programming, sequencing plays a crucial role. Proper sequencing ensures that events are handled and processed in the expected order, allowing the code to respond appropriately. 6. Data Processing: Sequencing is used when processing data or performing calculations. The order in which operations are performed can significantly impact the accuracy and efficiency of the code. By sequencing mathematical or logical operations correctly, developers can achieve the desired results. 7. Code Organization: Sequencing helps in organizing code and improving code readability. By following a logical sequence, it becomes easier to understand the flow and structure of the program. This is particularly important when working on collaborative projects or when revisiting code after a long time. Overall, sequencing is a fundamental aspect of coding that provides structure, logic, and control to programs. By properly arranging and sequencing instructions, developers can create efficient and functional code that achieves the desired outcome. Examples of sequencing in coding Sequencing is an essential concept in coding that is used in various programming languages and scenarios. Here are a few examples that illustrate how sequencing is used in coding: 1. Basic Arithmetic Operations: In a simple arithmetic operation, such as adding two numbers, sequencing is crucial. For example, in Python: num1 = 5 num2 = 3 sum = num1 + num2 The sequencing of instructions ensures that the values of `num1` and `num2` are retrieved from memory and added together before the result is stored in the variable `sum`. 2. Iterations: Sequencing is used in loops to repeat a set of instructions until a certain condition is met. In JavaScript, for example: javascript for (let i = 1; i = 18) { System.out.println("You are an adult."); } else { System.out.println("You are a minor."); } The sequence of instructions within the conditional statement determines whether the message "You are an adult" or "You are a minor" is printed, depending on the value of the `age` variable. 4. HTML Structure: Sequencing is used in HTML coding to structure the elements of a web page. For example: html Here is some content... The sequencing of HTML tags ensures that the elements are nested and displayed in the correct order, creating the desired structure of the web page. These are just a few examples of how sequencing is used in coding. Whether it's performing mathematical operations, implementing loops, or structuring HTML elements, proper sequencing is crucial for achieving the desired functionality and outcome in coding. Tips for effective sequencing in coding Sequencing plays a vital role in coding, ensuring that instructions are executed in the correct order to achieve the desired outcome. Here are some tips for effective sequencing in coding: 1. Plan and Outline: Before diving into writing code, take the time to plan and outline the sequence of steps required to accomplish the task. Having a clear understanding of the desired outcome and the logical flow of the program will help in organizing the code effectively. 2. Break Down Tasks: Divide complex tasks into smaller, manageable steps. Sequencing becomes more manageable when each step is clear and defined. This will improve code readability and allow for easier debugging or modification in the future. 3. Use Clear and Descriptive Names: When naming variables, functions, and classes, use descriptive names that reflect their purpose and role in the code. This makes it easier to understand the flow and purpose of the program and helps in maintaining a logical sequencing structure. 4. Follow Programming Guidelines: Different programming languages have their own guidelines and best practices. Familiarize yourself with these guidelines, including coding conventions, indentation, and comment usage. Adhering to these guidelines improves clarity and helps maintain a consistent sequencing pattern throughout the code. 5. Test and Validate: After writing the code, test it thoroughly to ensure that the sequencing produces the expected results. Validate the output at each step and verify that the code functions as intended. This helps identify any potential errors or logical flaws in the sequencing. 6. Document Your Code: Documenting your code is essential for yourself and other developers who may work on the project. Clearly explain the purpose and sequence of each section of the code through comments and documentation. This ensures that the sequencing is easily understandable and can be modified or maintained effectively. 7. Review and Refactor: Regularly review your code, looking for any opportunities to refactor or optimize the sequencing. This could involve simplifying complex sequences, removing redundancies, or restructuring code for better readability and efficiency. 8. Utilize Version Control: Version control systems, such as Git, allow you to track changes in your code and collaborate with others. Use version control to manage different versions of your codebase, making it easier to roll back changes or collaborate on sequencing improvements. By incorporating these tips into your coding practice, you can ensure effective sequencing and create well-structured, reliable, and maintainable code. Common challenges in sequencing and how to overcome them While sequencing is crucial in coding, it can also pose certain challenges that developers may encounter. Here are some common challenges in sequencing and strategies to overcome them: 1. Logical Errors: One of the most common challenges is encountering logical errors in the sequencing of code. This can result in unexpected outcomes or program failures. To overcome this, it is important to thoroughly plan and review the sequencing before writing the code. Writing pseudocode or using flowcharts can help visualize the sequence and identify potential logic flaws. 2. Race Conditions: In multi-threaded or concurrent programming, race conditions can occur when multiple threads access and modify shared resources simultaneously. This can lead to unpredictable results or data corruption. To address race conditions, developers can use synchronization mechanisms, such as locks or semaphores, to control access and ensure proper sequencing of actions. 3. Dependencies: Sometimes, the sequencing of code is dependent on external resources or other parts of the program. Managing dependencies can be challenging, especially when certain actions need to be executed in a specific order. One way to overcome this challenge is to use callback functions, promises, or asynchronous programming techniques that allow for non-blocking code execution and proper sequencing of dependent actions. 4. Inefficient Sequencing: Poorly structured or inefficient sequencing can result in slower execution and reduced performance. To overcome this challenge, it is crucial to analyze the code and identify areas where optimization is possible. This may involve reordering actions, eliminating unnecessary repetitions, or utilizing more efficient algorithms to improve the overall sequencing and execution time. 5. Handling Errors and Exceptions: Error handling and exception management can be complex, especially when dealing with multiple potential errors and unexpected scenarios. To handle errors effectively, developers can implement try-catch blocks or use exception handling mechanisms provided by the programming language. Properly sequencing error handling code allows for graceful recovery or appropriate error messages. 6. Complex Control Flow: When dealing with complex control flow structures, such as nested loops and conditionals, it can be challenging to ensure the correct sequencing of actions and conditions. To overcome this, it is essential to carefully analyze the control flow and use clear indentation, comments, and logical structuring of the code. Breaking down complex sequences into smaller, manageable functions or methods can also improve readability and maintainability. 7. Debugging and Testing: Debugging and testing can become challenging when the sequencing of code is not evident or when errors are not easily reproducible. To address this, developers can utilize debugging tools provided by their coding environment, use logging statements to trace the execution path, and create comprehensive unit tests that cover different scenarios to ensure proper sequencing and identify any issues. By being aware of these common challenges and employing the appropriate strategies, developers can overcome sequencing hurdles, create efficient code, and ensure that their programs function as intended. Conclusion Sequencing is a fundamental concept in coding that determines the order in which instructions are executed. It plays a crucial role in creating well-structured, functional, and efficient programs. By ensuring that code is organized and executed in the right sequence, developers can achieve the desired outcomes and streamline the flow of their applications. In this article, we explored the definition of sequencing in coding and why it is important. We discussed how sequencing is used in various coding scenarios, such as arithmetic operations, loops, conditional statements, and HTML structure. We also provided tips for effective sequencing, including planning and outlining, breaking down tasks, following programming guidelines, and testing and validating the code. However, it's important to recognize that sequencing in coding is not without its challenges. Logical errors, race conditions, dependencies, inefficient sequencing, complex control flow, and debugging complexities can all pose hurdles. By being aware of these challenges and utilizing strategies to overcome them, developers can mitigate risks and create more robust and reliable code. Ultimately, mastering sequencing is essential for any coder seeking to build high-performing and functional applications. The ability to arrange instructions in the correct order, manage dependencies, and handle complex control flow will contribute to code that is easier to read, maintain, and troubleshoot. As you continue your coding journey, remember to pay close attention to sequencing. Embrace best practices, adhere to coding guidelines, and continuously improve your skills in planning, organizing, and executing code in the right sequence. By doing so, you'll become a more proficient coder capable of crafting elegant and efficient solutions. The sequence is described as a systematic collection of numbers or events called as terms, which are arranged in a definite order. Arithmetic and Geometric sequences are the two types of sequences that follow a pattern, describing how things follow each other. When there is a constant difference between consecutive terms, the sequence is said to be an arithmetic sequence, On the other hand, if the consecutive terms are in a constant ratio, the sequence is geometric. In an arithmetic sequence, the terms can be obtained by adding or subtracting a constant to the preceding term, wherein in case of geometric progression each term is obtained by multiplying or dividing a constant to the preceding term. Here, in this article we are going to discuss the significant differences between arithmetic and geometric sequence. Content: Arithmetic Sequence Vs Geometric Sequence Comparison Chart Definition Key Differences Conclusion Comparison Chart Basis for ComparisonArithmetic SequenceGeometric Sequence MeaningArithmetic Sequence is described as a list of numbers, in which each new term differs from a preceding term by a constant quantity.Geometric Sequence is a set of numbers wherein each element after the first is obtained by multiplying the preceding number by a constant factor. IdentificationCommon Difference between successive terms.Common Ratio between successive terms. Advanced byAddition or SubtractionMultiplication or Division Variation of termsLinearExponential Infinite sequencesDivergentDivergent or Convergent Definition of Arithmetic Sequence Arithmetic Sequence refers to a list of numbers, in which the difference between successive terms is constant. To put simply, in an arithmetic progression, we add or subtract a fixed, non-zero number, each time infinitely. If a is the first member of the sequence, then it can be written as: a, a+d, a+2d, a+3d, a+4d.. where, a = the first term d = common difference between terms Example: 1, 3, 5, 7, 9… 5, 8, 11, 14, 17… Definition of Geometric Sequence In mathematics, the geometric sequence is a collection of numbers in which each term of the progression is a constant multiple of the previous term. In finer terms, the sequence in which we multiply or divide a fixed, non-zero number, each time infinitely, then the progression is said to be geometric. Further, if a is the first element of the sequence, then it can be expressed as: a, ar, ar2, ar3, ar 4 … where, a = first term d = common difference between terms Example: 3, 9, 27, 81… 4, 16, 64, 256.. There are many, many programming languages available that allow us to program computers to solve all kinds of problems. There are scripting languages, systems languages, web programming languages, dynamic languages, object-oriented languages, functional languages, and the list goes on and on. But did you know that all programming languages have 3 elements in common? Three very simple elements that give us the power to implement solutions to extremely complex problems. These 3 elements are: Sequence Selection Iteration Sure, many programming languages have many other complex features. Some are 'easy' to learn and others more difficult to learn. In this post I'd like to talk about each one of these elements and build a very simple C++ program that uses all of them. C++ is one of those languages that is considered very difficult to learn because it is very complex. Let's talk about each of these elements individually and we'll write a simple C++ program along the way that uses all of them. We'll keep this example simple and I'm sure you will be able to follow along. If you'd like to follow along by typing or copying/pasting the code below, you can do so without installing any software at all. Simply point your favorite browser to � and select C++17 from the dropdown list at the upper right.Then delete the text in the online editor window and type or copy/paste the code we'll write along the way. When you are ready to run the program, simply click on the green Run button at the top of the screen. If you see any errors, then double check that you entered the code exactly as shown and try it again. Once the program runs, you will be able to enter data and see output at the bottom of the screen.So, what are these 3 elements all about?It's actually very simple. In order to solve problems with any programming language, we write code that tells the computer what operations to execute and in what order. The order must be very specific – remember the computer is not very smart – it simply follows our instructions. These operations make up what is called an algorithm. Just a fancy word that describes a set of operations that solves a specific problem. You can think of this very much like a cooking recipe. If you follow the recipe exactly, you will end up with the produce of that recipe.Sequence, Selection, and Iteration are the basic elements that we use to tell the computer what to do. The code will definitely look different depending on the programming language we use, but the algorithm will be the same.So let's describe these elements:Sequence– the order we want the computer to execute the instructions we provide as programmers. For example, do this first, then do this, then do that, and so forth.Selection– selecting which path of an algorithm to execute depending on some criteria. For example, if you passed a class in school, then we execute the operations that clap and cheer and play a song. But if you didn't pass the class, then maybe we would say, "Better luck next time, hang in there!"Iteration– looping or repeating. Many times, we want to be able to repeat a set of operations a specific number of times or until some condition occurs.That's it, these 3 super simple elements give us the ability to write programs that solve problems. When we put them together we can create programs that are very complex such as operating systems, game engines, compilers, anything! In fact, with just Sequence, Selection, and Iteration we can implement any algorithm.Read that again! Any algorithm! That's a very powerful place to be!!Alright, let's write some C++ code together.Sequence Let's start with Sequence. Most programming languages simply execute instructions one after another as they are read – much like reading a recipe or a book. Here's a simple C++ program that prompts the user to enter their age and then reads what they type in on the keyboard into a variable and then displays "Bye." to the display console. #include using namespace std; int main() { int age {0}; cout > age; cout age; if (age >= 18) cout