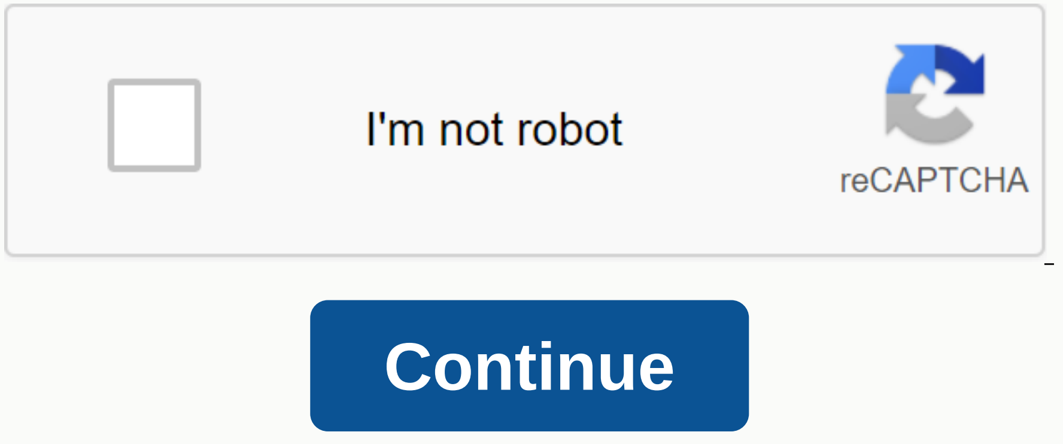
□ I'm not robot
reCAPTCHA
Continue

Python print list vertically

In this notebook, you will learn to store more than one valuable variable. This alone is one of the most powerful ideas in programming, and it introduces a number of other central concepts such as loops. If this section makes sense to you, you can write some interesting programs, and you can be more confident that you will be able to develop overall programming skills. A list is a collection of items that is stored in a variable. The items should be linked in some way, but there are no restrictions on what can be stored in a list. Here is a simple example of a list, and how we can quickly access each item in the list. Students = ['bernice', 'aaron', 'cody'] for students in students: print(Hello, + student.title() + !) Hello, Bernice! Hello, Aaron! Hello, Cody! Because lists are collections of objects, it makes sense to give them a plural name. If each item in your list is a car, call the Cars list. If each item is a dog, call your list dogs. This gives you an easy way to point to the entire list (dogs) and to a single item in the list (dog). In Python, square brackets denote a list. To define a list, type the name of the list, the equal sign, and the values you want to add to the list in square brackets. Dogs = ['Border Collie', 'Australian Cattle Dog', 'labrador retriever'] items in a list are identified by their position in the list, starting with zero. That will almost certainly rise at some point. Programmers even joke about how often we all make off-by-one mistakes, so don't feel bad if you make these kinds of mistakes. To access the first item in a list, specify the name of the list, followed by a zero in parentheses. dogs = ['border collie', 'australian cattle dog', 'labrador retriever'] dog = dogs[0] print(dog.title()) The number in brackets is called the index of the article. Because lists start from zero, an item's index is always one smaller than its position in the list. So to get the second item in the list, we need to use an index of 1. dogs = ['border collie', 'australian cattle dog', 'labrador retriever'] dog = dogs[1] print(dog.title()) You can probably see that we would use an index of 2 to get the last item in this list. This works, but it would only work because our list contains exactly three elements. To get the last item in a list, no matter how long the list is, you can use an index of -1. dogs = ['border collie', cattle dog', 'labrador retriever'] dog = dogs[-1] print(dog.title()) This syntax also works for the penultimate point, the third last point and so on. Dogs = ['border collie', 'Australian cattle dog', 'labrador retriever'] dog = dogs[-2] print(dog.title()) However, you cannot use a negative number that is larger than the length of the list. Dogs = ['Border Collie', 'Australian cattle dog', 'labrador retriever'] retriever'] = dogs[-4] print(dog.title()) --- IndexError Traceback (last call) &lt;ipython-input-33-32c58df001ad&gt;in &lt;module&gt;() 1 dogs = ['border collie', 'australian cattle dog', 'labrador retriever'] 2 ----&gt; 3 dog = dogs[-4] 4 print(dog.title()) IndexError: list index out of range Exercises'First List' Save the values 'python', 'c', print each of these values using their position in the list. First decent list- Save the values 'python', 'c' and 'java' in a list. Print a statement on each of these values using their position in the list. Your statement could be simple: A beautiful programming language is worth it. Think of something you can save in a list. Create a list of three or four items, and then print a message that contains at least one item from the list. Your sentence could be as simple as: An item in my list is a __. This is one of the most important concepts related to lists. You can include a list of one million articles, and you can write a sentence for each of these million articles in three lines of code. If you want to understand lists and become a competent programmer, you should take the time to understand this section. We use a loop to access all items in a list. A loop is a block of code that repeats until no more items are executed for work or until a specific condition is met. In this case, our loop is executed once for each point in our list. With a list that is three points long, our loop runs three times. Let's take a look at how we access all the items in a list, and then try to understand how it works. Dogs = ['Border Collie', 'Australian Cattle Dog', 'Labrador Retriever'] for dog in dogs: print(dog) border collie australian cattle dog labrador retriever We've already seen how to create a list, so we're really just trying to understand how the last two lines work. These last two lines form a loop, and the language here can help us see what's happening: for dogs: The keyword for says Python to prepare for using a loop. The variable dog, without s on it, is a temporary placeholder variable. This is the variable in which Python places each item in the list, one at a time. The first time through the loop will be the value of dog Border Collie. The second time through the loop will be the value of dog Australian cattle dog. The third time through, dog will be Labrador retriever. After that, there are no items more in the list, and the loop is terminated. You can use the pythontutor.com Web site to run Python code one line at a time. As you run the code, there is also a visualization on the screen that shows you how the dog variable contains different values over the course of the loop. There is also an arrow that moves around your code and shows you how to use some lines&lt;/module&gt; &lt;/ipython-input-33-32c58df001ad&gt; &lt;/ipython-input-33-32c58df001ad&gt; only once, while other lines run multiple tiimes. To see this in action, click the Forward button and view the visualization and output as it prints on the screen. Tools like this are incredibly valuable to see what Python does with your code. We can do with the value dog within the loop what we want. In this case, we only print the dog's name. print(dog) We are not limited to printing the word dog. We can do whatever we want with this value, and this action is carried out for every item in the list. Let's say something about each dog in our list. Dogs = ['Border Collie', 'Australian Cattle Dog', 'labrador retriever'] for dogs: print('I like' + dog + 's.') I like Border Collies. I like Australian cattle dogs. I like Labrador Retriever. Visualize this on pythontutor. Python uses indent to decide what is inside the loop and what is outside the loop. Code that is inside the loop runs for each item in the list. Code that is not indented, that comes after the loop, runs once like regular code. dogs = ['border collie', 'australian cattle dog', 'labrador retriever'] for dog in dogs: print('I like ' + dog + 's.') print('No, I really like ' + dog +'s!') print(That's just how I feel about dogs.) I like Border Collies. No, I really like Border Collies! I like Australian cattle dogs. No, I really like Australian cattle dogs! I like Labrador Retriever. No, I really like Labrador Retriever! That's how I feel about dogs. Note that the last row runs only once after the loop completes. Also note the use of line lines () to make the output easier to read. Run this code on pythontutor. When you loop through a list, you should know the index of the current item. You can always use the List.index(value) syntax, but there is an easier way. The Enumerate() function tracks the index of each item for you as it drags through the list: dogs = ['border collie', 'australian cattle dog', 'labrador retriever'] print(Results for the dog show are as follows:) for Index, Dog in enumerate(dogs): place = str(index) print(Place: + place + Dog: + dog.title()) Results for the dog show are as follows: Place: 0 Dog: Border Collie : 1 Dog: Australian Cattle Dog Place: 2 Dog: Labrador Retriever To list a list, you need to add an index variable to keep the current index. So instead of for dogs in dogs: you have for index, dog in enumerating(dogs) The value in the variable index is always a whole If you want to print it in a string, you need to convert the integer to a string: str(index) The index always starts at 0, so in this example the value of the place should actually be the current index, plus one: Dogs = ['Border Collie', 'Australian Cattle Dog', 'labrador retriever'] print(Results print(Results print(Results print the dog show are as follows:) for index, dog in enumerate(dogs): place = str(index + 1) print(Place: + place + Dog: + dog.title()) Results for the dog show are as follows: Place: 1 Dog: Border Collie Place: 2 Dog: Australian Cattle Dog Place: 3 Dog: Labrador Retriever A frequent looping error occurs when instead of using the individual variable dog in the loop, We use the variable : Dogs = ['Border Collie', 'Australian Cattle Dog', 'labrador retriever'] for dogs in dogs: print(dogs) ['border collie', 'australian cattle dog', 'labrador retriever'] ['border collie', 'Australian cattle dog', 'labrador retriever'] ['border collie', 'australian cattle dog', 'labrador retriever'] In this example, we do not print every dog in the list, but print the entire list every time we print the whole list. Python inserts every single item in the list into the dog variable, but we never use this variable. Sometimes you only get a mistake when you try to do this: dogs = ['Border Collie', 'Australian cattle dog', 'labrador retriever'] for dogs in dogs: print('I like' + dogs + 's.') --- TypeError Traceback (last call last) &lt;ipython-input-20-8e7acc74d7a9&gt;in &lt;module&gt;() 2 3 for dogs in dogs: ----&gt; 4 print('I like' + dogs + 's.') TypeError: Can't convert 'list' objekt to str implicit Exercises'First List - Loop. Repeat First List, but this time use a loop to print out each value. First Decent List - Loop - Repeat the first neat list, but this time use a loop to print your instructions. Make sure you write the same set for all the values in your list. Loops are not effective when you try to generate different outputs for each value in The List. Your First List - Loop - Repeat your first list, but this time use a loop to print your message for each item in your list. If you have created different messages for each value in your list, choose a message to repeat for each value in your list. You can change the value of any item in a list if you know the position of that item. Dogs = ['border collie', 'australian cattle dog', 'labrador retriever'] dogs[0] = 'australian shepherd' print(dogs) ['australian shepherd', 'australian cattle dog', 'labrador retriever'] If you want to find out the position of an item in a list, you can use the Index() function. dogs = ['border collie', 'australian cattle dog', 'labrador retriever'] print(dogs.index('australian cattle dog')) This method gives a if the requested item is not in the list. dogs = ['border collie', 'australian cattle dog', 'labrador retriever'] print(dogs.index('pudel')) --- ValueError Traceback (last call last) &lt;ipython-input-13-a9e05e37e8df&gt;in &lt;module&gt;() 1&lt;/module&gt; &lt;/ipython-input-13-a9e05e37e8df&gt; &lt;/module&gt; &lt;/ipython-input-20-8e7acc74d7a9&gt; &lt;/ipython-input-20-8e7acc74d7a9&gt; = ['border collie', 'australian cattle dog', 'labrador retriever'] 2 ----&gt; 3 print(dogs.index('pudel')) ValueError: 'pudel' is not in the list You can test whether an item is in a list with the keyword in . This becomes more useful after you learn how to use if-else statements. Dogs = ['Border Collie', 'Australian Cattle Dog', 'labrador retriever'] print('australian cattle dog' in dogs) print('pudel' in dogs) We can add an article to a list using the Append() method. This method adds the new item to the end of the list. Dogs = ['border collie', 'australian cattle dog', 'labrador retriever'] dogs.append('pudel') for dogs: print(dog.title() + s are cool.) Border Collies are cool. Australian Cattle Dogs are cool. Labrador retrievers are cool. Poodles are cool. We can also insert elements wherever we want, using the insert() function. We define the position that the point should have, and everything that is from that point is moved one position to the right. In other words, the index of each element after the new element is incremented by one. dogs = ['border collie', 'australian cattle dog', 'labrador retriever'] dogs.insert(1, 'poodle') print(dogs) ['border collie', 'poodle', 'australian cattle dog', 'labrador retriever'] Note that you must first enter the position of the new element and then the value of the new item. If you do this in reverse order, you will receive an error. Now that we know how to add items to a list after it's created, we can use lists more dynamically. We are no longer firmly away from defining our entire list at once. A common approach to lists is to define an empty list and then have the program add items to the list as needed. For example, this approach works when creating an interactive Web site. Your user list may start blank, and as users sign up for the site, it will grow. This is a simplified approach to how websites actually work, but the idea is realistic. Here's a quick example of how to start with an empty list, start filling in, and work with the items in the list. The only new thing here is the way we define an empty list that is just an empty set of square brackets. • Create an empty list to keep our users. User names = [] - Add some users. usernames.append('bernice') usernames.append('cody') usernames.append('aaron') for usernames in username: print(Welcome, + username.title() + '!') Welcome, Bernice! Welcome, Cody! Welcome, Aaron! If we do not change the order in our list, we can use the list to who our oldest and newest users are. • Create an empty list to keep our users. User names = [] - Add some users. usernames.append('bernice') usernames.append('cody') usernames.append('aaron') for usernames in username: print(Welcome, + username.title() + '!') • Recognize our first user and welcome our newest user. User. you as our very first user, + username[0].title() + '!') print(And welcome to our newest user, + username[-1].title() + '!') Welcome, Bernice! Welcome, Cody! Welcome, Aaron! Thank you for being our very first user, Bernice! And welcome to our newest user Aaron! Note that the code that welcomes our newest user always works because we used the index -1. If we had used Index 2, we would always get the third user, even as our list of users grows and grows. We can sort a list alphabetically in one of the two order. students = ['bernice', 'aaron', 'cody'] - Put the students in alphabetical order. students.sort() - Displays the list in the current order. print(Our students are currently in alphabetical order.) for students: print(student.title()) #Put students in reverse alphabetical order. students.sort(reverse=True) - View the list in their current order. print(Our students are now in reverse alphabetical order.) for students in students: print(student.title()) Our students are currently in alphabetical order. Aaron Bernice Cody Our students are now in reverse alphabetical order. Cody Bernice Aaron If you are considering sorting a list, remember that you cannot restore the original order. If you want to display a list in sorted order, but want to keep the original order, you can use the sorted() function. The sorted() function also accepts the optional reverse=True argument. students = ['bernice', 'aaron', 'cody'] - Show students in alphabetical order, but keep the original order. print(Here is the list in alphabetical order:) for students in sorted (students): print(student.title()) - Show students in reverse alphabetical order, but keep the original order. print(Here is the list in reverse alphabetical order:) for students in sorted (students, reverse=True): print(student.title()) print(Here is the list in the original order:) - Show, that the list is still in the original order. for students in students: print(student.title()) Here is the list in alphabetical order: Aaron Bernice Cody Here is the list in reverse alphabetical order: Cody Bernice Aaron Here is the list in the original order: Bernice Aaron Cody We have seen three possible jobs for a list: The original order in which the list was created alphabetical order. Order of the original order. The reverse() function gives us this order. students = ['bernice', 'aaron', 'cody'] students.reverse() print(students) ['cody', 'aaron', 'bernice'] Note that the back is permanent, although you could track another call to reverse() and get the original order of the list back. All sorting functions also work for numeric lists. pay = [1, 3, 4, 2] - sort() sets numbers in increasing order. numbers.sort() print(numbers) - sort(reverse=True) sets in decreasing order. numbers.sort(reverse=True) print(numbers) [1, 2, 3, 4] [4, 3, 2, 1] pay = [1, 3, 4, 2] - sorted() keeps the original order of the list: print(sorted(numbers)) print(numbers) [1, 2, 3, 4] [1, 3, 4, 2] numbers = [1, 3, 4, 2] numbers.reverse() print(numbers) You can find the length of a list with the function len(). Username = ['bernice', 'cody', 'aaron'] user_count = len(username) print(user_count) There are many situations where you might want to know how many items are in a list. If you have a list that stores your users, you can find the length of the list at any time and know how many users you have. • Create an empty list to keep our users. User Name = [] - Add some users and report how many users we have. usernames.append('bernice') user_count = len(username) print(We have + str(user_count) + user!) usernames.append('cody') usernames.append('aaron') user_count = len(usernames) print(We have + str(user_count) + users!) We have 1 user! We have 3 users! Technically, the len() function returns an integer that cannot be printed directly with strings. We use the str() function to turn the integer into a string so that it is beautifully printed: username = ['bernice', 'cody', 'aaron'] user_count = len(usernames) print(This causes an error: + user_count) --- TypeError Traceback (last call last) &lt;ipython-input-43-92e732ef190e&gt;in &lt;module&gt;() 2 user_count = len(username) 3 ----&gt; 4 print(This causes an error: + user_count) TypeError : Can't convert 'int' object to str implicitly usernames = 'aaron'] user_count = len(username) print(This works: + str(user_count)) exercises, worklist, Create a list that contains four careers, such as 'programmer' and 'truck driver'. Use the list.index() function to find the index of a career in your list. Use the In function to indicate that this career is on your list. Use the append() feature to add a new career to your list. Use the insert() function to add a new career to the top of the list. Use a loop to see all the careers in your list. Create the list that landed you in the worklist, but this time start the file with an empty list and fill it out with Append() statements. Print a statement that tells us what the first career you thought was. Print a statement that tells us what the last career you thought was. Ordered Start with the list that you created in the work list. You will print the list in different jobs. Each time you print the list, use a for loop instead of printing the raw list. Print a message each time in which it tells us in which order you want the list to appear. Print the list in the original order. Print the list in&lt;/module&gt; &lt;/ipython-input-43-92e732ef190e&gt; &lt;/ipython-input-43-92e732ef190e&gt; Order. Print the list in the original order. Print the list in reverse alphabetical order. Print the list in the original order. Print the list in reverse order from what it started. Print the list in its original order Permanently sort the list in alphabetical order, and then print it. Sort the list permanently in reverse alphabetical order, and then print it. Ordered Numbers- Create a list of 5 numbers in a random order. You will print the list in different jobs. Each time you print the list, use a for loop instead of printing the raw list. Print a message each time in which it tells us in which order you want the list to appear. Print the numbers in the original order. Print the numbers in increasing order. Print the numbers in the original order. Print the numbers in decreasing order. Print the numbers in their original order. Print the numbers in reverse order from the beginning. Print the numbers in the original order. Sort the numbers permanently in increasing order, and then print them. Sort the numbers permanently in debt-ridden order, and then print them. List Lengths - Copy two or three of the lists you created from the previous exercises, or create two or three new lists. Print out a series of statements that tell us how long each list is. Hopefully you can now see that lists are a dynamic structure. We can define an empty list and then fill it when information comes into our program. To become truly dynamic, we need some ways to remove items from a list when we no longer need them. You can remove items from a list from their location or by their value. If you know the position of an item in a list, you can remove that item with the del command. To use this approach, enter the del command and the name of your list with the index of the item you want to move in square brackets: dogs = ['border collie', 'australian cattle dog', 'labrador retriever'] - Remove the first dog from the list. del dogs[0] print(dogs) ['australian cattle dog', 'labrador retriever'] You can also remove an item from a list if you know its value. To do this, we use the remove() function. Specify the name of the list, followed by the Remove word with the value of the item you want to remove in parentheses. Python searches your list, finds first item with this value and removes it. dogs = ['border collie', 'australian cattle dog', 'labrador retriever'] ' Remove Australian cattle dog from the list. dogs.remove('australian cattle dog') print(dogs) ['border collie', 'labrador retriever'] However, make sure that only the first item with this value is removed. If you have multiple items with the same value, you have some items with that value in the list. Letters = ['a', 'b', 'c', 'a', 'b', 'c'] - Remove the letter a from the list. letters.remove('a') print(letters) ['b', 'c', 'a', 'b', 'c'] 'c'] is a cool concept in programming called popping elements from a collection. Each programming language has a kind of data structure that resembles Python's lists. All of these structures can be used as queues, and there are several ways to process the items in a queue. A simple approach is to start with an empty list and then add items to that list. If you want to work with the items in the list, always remove the last item from the list, do something with it, and then remove that item. The pop() function makes this easy. It removes the last item from the list and gives it to us so we can work with it. This is easier to show with an example: Dogs = ['Border Collie', 'Australian Cattle Dog', 'labrador retriever'] last_dog = dogs.pop() print(last_dog) print(dogs) labrador retriever ['border collie', 'australian cattle dog'] This is an example of a first-in, last-out approach. The first item in the list would be the last item to be processed if you continue to use this approach. We will see a full implementation of this approach later when we learn about while loops. You can actually populate any item you want from a list by specifying the index of the item you want to suspend and suspend. So we could make a first-in,first-out approach by including the first iem in the list: Dogs = ['Border Collie', 'Australian Cattle Dog', 'labrador retriever'] first_dog = dogs.pop(0) print(first_dog) print(dogs) border collie ['australian cattle dog', 'labrador retriever'] Exercises'Famous People' Make a list that includes the names of four. Remove each person from the list, one by one, using each of the four methods we just saw: pop the last item from the list, and pop each item except the last item. Remove an item by its position and an item by its value. Print a message that there are no more famous people in your list, and print your list to prove it's empty. At this point, you may have noticed that in some of our examples we have some seimtive codes. This repetition disappears as soon as we learn how to use functions. If this replay already bothers you, you should look at introductory features before doing any further exercises in this section. Because a list is a collection of items, we should be able to get any subset of those items. For example, if we want to get only the first three points from the list, we should be able to do so easily. The same should be done for all three elements in the list, or for the last three items or for x items from anywhere in the list. These subsets of a list are called slices. To get a subset of a list, we specify the position of the first item we want and the position of the first item that we don't want to include in the subset. Thus, the slice list[0:3] returns a list of items 0, 1, and 2, but not item 3. Learn how to get a batch that contains the first three items. Username = = 'cody', 'aaron', 'ever', 'dalia'] - Grab the first three users in the list. first_batch = username[0:3] for users in first_batch: print(user.title())) If you want to pack everything up to a certain position in the list, you can also leave the first index blank: username = ['bernice', 'cody', 'aaron', 'ever', 'dalia'] - Grave the first three users in the list. first_batch = username[:3] for users in first_batch: print(user.title()) When we get a slice from a list, the original list is not affected: username = ['bernice', 'cody', 'aaron', 'ever', 'dalia'] - Grab the first three users in the list. first_batch = User Name[0:3] - The original list is not affected. for users in username: print(user.title()) Bernice Cody Aaron Ever Dalia We can get any segment of a list we want, using the slice method: username = ['bernice', 'cody', 'aaron', 'ever', 'dalia'] - Grab a stack from the middle of the list. middle_batch = username[1:4] for users in middle_batch: print(user.title())) To get all items from one position in the list to the end of the list, we can omit the second index: username = ['bernice', 'cody', 'aaron', 'ever', 'dalia'] - Grab all users from the third to the end. end_batch = user name[2:] for users in end_batch: print(user.title()) You can use slice notation to make a copy of a list by omitting both the start and end index. This will make the segment consist of everything from the first element to the last item, that is, the entire list. Username = ['bernice', 'cody', 'aaron', 'ever', 'dalia'] - Make a copy of the list. copied_usernames = User Name[:] print(The full copied list:'t'' copied_usernames) - Remove the first two users from the copied list. del copied_usernames[0] del copied_usernames[0] print(Removed two users from the copied list:'t,copied_usernames) - The original list is not affected. print(The original list:'t, Username) The full copied list: ['bernice', 'cody', 'aaron', 'ever', 'dalia'] Two users removed from the copied list: ['aaron', 'ever', 'dalia'] The original list: ['bernice', 'cody', 'aaron', 'ever', 'dalia'] Exercises'Alphabet Slices' Use a segment to print the first three letters of the alphabet. Use a segment to print any three letters from the middle of the list. Use a segment to print the letters from any point in the middle of the list to the end. Protected List- Your goal in this exercise is to prove that copying a protects the original list. Create a list of three names. Use a slice to make a copy of the entire list. Add at least two new names to the new copy of the list. Create a loop that outputs all the names in the original list, along with a message that this is the original list. Create a loop that outputs all the names in the copied list, along with a message that this is the copied list. There are special about lists of numbers, but there are some features that you can use to make working with numeric lists more efficient. Let's make a list of the first ten numbers and start to see how we can use numbers in a list. • Print the first ten numbers. pay = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] for numbers: print(number) This works, but it is not very efficient if we want to work with a large set of numbers. The Range() function helps us to generate long number lists. Here are two ways to do the same with the range function. • Print the first ten numbers. for number in range(1,11): print(number) The range function records a start number and an end number. You get all the integers, up to and including the final number. You can also add a step value that indicates to the range function how big a step between numbers should be: for number in range(1,21,2): print(number) If we want to store these numbers in a list, we can use the List() function. This function occupies a range and turns it into a list: pay = list(range(1,11)) print(numbers) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] This is incredibly powerful; we can now create a list of the first million numbers, just as we have created a list of the first ten numbers. It makes no sense to print the millions of numbers here, but we can show that the list really contains a million items, and we can print the last ten items to show that the list is correct. • Save the first million numbers to a list. numbers = list(range(1,1,1000001)) - Show the length of the list: print(The list 'numbers' has + str(len(numbers)) + numbers in it.) • Show the last ten numbers: print(The last ten numbers in the list are:) for numbers in numbers[-10:]: print(number) The 'Numbers' list has 1000000 numbers in it. The last ten numbers in the list are: 999991 999992 999993 999994 999995 999996 999997 9999999 9999999 1000000 The expression str(len(numbers)) takes the length of the number list and turns it into a string that can be printed. The expression numbers[-10:] gives us a part of the list. The index -1 is the last item in the list, and the index -10 is the item ten places from the end of the list. So the slice numbers[-10:] give us everything from this element to the end of the list. There are three functions that you can easily use with numeric lists. As you might expect, the min() function returns the smallest number in the list, the max() function returns the largest number in the list, and the Sum() function returns the sum of all the numbers in the list. Age = [23, 16, 14, 28, 11, 38] youngest = min(alter) oldest = max(ages) total_years = sum(ages) print(Our youngest reader is + str(youngest) + years old.) print(Our oldest oldest is + str(oldest) + years old.) print(Together we have + str(total_years) + years of life experience.) Our youngest reader is 11 years old. Our oldest reader is 38 years old. Together we have 149 years of life experience. Exercises-First Twenties Use the range() function to save and print the first twenty numbers (1-20) in a list. Larger Sets- Take the first_twenty.py program you just wrote. Change your final number to a much larger number. How long does it take for your computer to print the first millions of numbers? (Most people will never see a million numbers scrolling in front of their eyes. You can see that now!) Five wallets - Imagine five wallets with different amounts of cash in them. Save these five values in a list and print the following sentences: The thickest wallet has a value of 'value'. The simplest wallet has a value of value. Overall, these wallets have a value of value. I thought carefully before this section was included. If you're new to programming, list understandings can look confusing at first glance. They are a short-lived way to create and work with lists. It's good to be aware of the list understandings because you'll see them in other people's code, and they're really useful when you understand how they're used. That is, if they don't make sense to you yet, don't worry about using them right away. If you have worked with enough lists, you should use understandings. At the moment it is good enough to know that they exist and to recognize them when you see them. If you like them, go ahead and start using them now. Numerical refinements.Let's look at how we could create a list of the first ten square numbers. We could do it like this: Save the first ten square numbers in a list. • Create an empty list that contains our square numbers. squares = [] - Go through the first ten numbers, square them, and add them to our list. for number in range(1,11): new_square = number**2 squares.append(new_square) - Indicate that our list is correct. for squares in squares: print(square) 1 4 9 16 25 36 49 64 81 100 This should be useful at this point. If this is not the case, go with these thoughts about the code: We create an empty list called squares that contain the values we are interested in. With the range() function, we start a loop that goes through the numbers 1-10. Each time we pass the loop, we find the square of the current number by placing it on the second Increase. We'll add this new value to our list squares. We go through our newly defined list and print out each square. Let's make this code more efficient. We don't really need to store the new square in a separate variable new_square; we can simply add it directly to the list of squares. The line new_square = number**2 is taken out, and the next line takes care of the quadrature: an empty list containing our square numbers. squares = [] - Go through the first ten numbers, square them, and add them to our list. for number in range(1,11): squares.append(Number**2) - Indicate that our list is correct. for squares in squares: print(square) 1 4 9 16 25 36 49 64 81 100 list understandings allow us to combine the first three lines of code into one line. Here's what it looks like: - Save the first ten square numbers in a list. squares = [number**2 for number in range(1,11)] - Indicate that our list is correct. for squares in squares: print(square) 1 4 9 16 25 36 49 64 81 100 It should be fairly clear that this code is more efficient than our previous approach, but it cannot be clear what is happening. Let's take a look at everything that happens in this first line: we define a list called squares. Look at the second part of what is in square brackets: for number in range(1,11) This sets up a loop that goes through the numbers 1-10 and stores each value in the variable number. Now we can see what happens to each number in the loop: number**2 Each number is incremented to the second power, and this is the value stored in the list we have defined. We could read this line as follows: quadrats = [Increase number to the second power, for each number in the range 1-10] It is probably helpful to see a few more examples of how comprehensibility can be used. Let's try to make the first ten straight numbers, the longer: . Make an empty list that keeps the even numbers. evens = [] - Loop through the numbers 1-10, double each, and add it to our list. for number in range(1,11): evens.append(number*2) - Show, that our list is correct: for straight: print(even) 2 4 6 8 10 12 14 16 18 20 So we could think the same, with a list understanding: straight = [multiply each number by 2, for each number in the range 1-10] Here is the same line in the code: straight = [number*2 for number in range(1,11)] for straight : print(even) 2 4 6 8 10 12 14 16 18 20 We can also use co-understanding with non-numeric lists. In this case, we'll create a first list and then use an understanding to create a second list from the first list. Here is a simple example, without understanding to use: studenten = ['bernice', 'aaron', 'cody'] - Let's turn them into great students. great_students = [] for students in students: great_students.append(student.title() + the big ones!) - Let's welcome every great student. for great_student in great_students: print(Hello, + great_student) Hello, Bernice the big one! Hello, Aaron the Hello, Cody the Great! To use an understanding in this code, let's write something like this: great_students = [add the big one to each student, for each student in the student list] This is what it looks like: = ['bernice', 'aaron', 'cody'] - Let's turn them into great students. great_students = [student.title() + the great! for students in students] - Let's welcome every great student. for great_student in great_students: print(Hello, + great_student) Hello, Bernice the big one! Hello, Aaron the Great! Hello, Cody the Great! Exercises. If these examples are useful, follow these steps and try to do the following exercises with understanding. If not, try the exercises without understanding. You can find out how to use understandings after you have solved each exercise in a longer way. Multiples of Tena Create a list of the first ten multiples of ten (10, 20, 30... 90, 100). There are a number of ways to do this, but try to do it with a list understanding. Print your list. We have seen how to create a list of the first ten squares. Create and print a list of the first ten cubes (1, 8, 27... 1000) with a list understanding. Awesomeness- Save five names in a list. Create a second list that is awesome to each name's expression! with a list understanding. Print the great version of the names. Work backwards- Write out the following code without using a list understanding: plus_thirteen = [Number + 13 for number in range(1,11)] Since you are now familiar with lists, we can take a second look at strings. A string is really a list of characters, so many of the concepts from working with lists behave the same with strings. We can go through a string with a for loop, just like we go through a list: message = Hello! for Letter in Message: print(letter) We can create a list from a string. The list has an element for each character in the string: message = Hello world! message_list = list(message) print(message_list) ['H', 'e', 'l', 'l', 'o', ',' 'w', 'o', 'r', 'l', 'd', '!'] We can access any character in a string by its position, just like we access individual items in a list: message = Hello World! first_char = message[0] last_char = message[-1] print(first_char, last_char) We can extend this to take slices of a string: message = Hello World! first_three = message[:3] last_three = message[-3:] print(first_three, last_three) A substring is a series of characters that are displayed in a string. You can use the keyword in to find out if a particular substring appears in a string: message = I like cats and dogs. dog_present = 'dog' in message print(dog_present) If you want to know where a substring in a string you can use the find() method. The find() method specifies the index at which the substring begins. message = I like cats and dogs. dog_index = message.find('dog') print(dog_index) Note, however, that this function returns only the index of the first appearance of the substring you are looking for. If the appears more than once, they will miss the other substrings. Message = I like cats and dogs, but I would much rather own a dog. dog_index = message.find('dog') print(dog_index) If you want to find the last appearance of a substring, you can use the rfind() function: message = I like cats and dogs, but I would much rather own a dog. last_dog_index = message.rfind('dog') print(last_dog_index) You can use the replacement function to replace a substring. To use the replace() function, type the substring that you want to replace, and then type the substring that you want to replace. You must also store the new string, either in the same string variable or in a new variable. message = I like cats and dogs, but I would much rather own a dog. message = message.replace('dog', 'snake') print(message) I like cats and snakes, but I would much rather own a snake. If you want to know how often a substring appears within a string, you can use the count() method. message = I like cats and dogs, but I would much rather own a dog. number_dogs = message.count('dog') print(number_dogs) strings can be split into a set of substrings if separated by a repeating character. If a string consists of a simple sentence, the string can be split based on spaces. The split() function returns a list of substrings. The split() function accepts an argument, the character that separates the parts of the string. message = I like cats and dogs, but I would much rather own a dog. words = message.split(' ') print(words) ['I', 'like', 'cats', 'and', 'dogs,', 'but', 'I'd', 'much', 'rather', 'own', 'a', 'dog.'] Note that punctuation remains in the substrings. It is more common to share strings that are really lists, separated by something like a comma. The split() function provides an easy way to convert comma-separated strings that you can't do much with in Python into lists. Once you have your data in a list, you can work with it in a much more powerful way. animals = dog, cat, tiger, mouse, liger, bear - Rewrite the string as a list and store it in the same variable animals = animals.split(',') print(animals) ['dog', 'cat', 'tiger', 'mouse', ' liger', ' bear'] Note that in this case the rooms are also ignored. It is a good idea to test the output of the split() function and make sure that it does what you want with the data you are interested in. A use is to work with spreadsheet data in your Python programs. Most spreadsheet applications allow you to store your data in a comma-separated text file. You can read this file in your Python program, or you can even copy it from the text file and paste it into your program file, and then convert the data to a list. You can then process your spreadsheet data using a for loop. There are a number of other string methods that we won't discuss here, but you should look at them. Most of these these should be useful for you at this point. You may not have any use for any of them at the moment, but it's good to know what you can do with strings. This way, you have a sense of how to solve certain problems, even if it means referring to the list of methods to remember how to write the correct syntax when you need it. Exercises-Listing a Sentence- Save a single set in a variable. Use a for loop to print each character from your sentence on a separate line. Set List- Save a single set in a variable. Create a list from your set. Print your raw list (don't use a loop, just print the list). Set Slices- Save a set in a variable. Use slices to print the first five characters, all five consecutive characters from the center of the sentence, and the last five characters of the sentence. When you search Python files, a set is stored in a variable to ensure that you use the word Python at least twice in the sentence. Use the keyword in to prove that the word Python is actually in the sentence. Use the find() function to show where the word Python appears first in the sentence. Use the rfind() function to see the last place where Python appears in the set. Use the count() function to see how often the word Python appears in your sentence. Use the split() function to divide your sentence into a list of words. Print the raw list and use a loop to print each word on its own line. Use the replace() function to change Python in your set to Ruby. ChallengesCounting DNA nucleotides project Rosalind is a problem set based on biotechnological concepts. It is intended to show how programming skills can help solve problems in genetics and biology. If you understand this section about strings, you have enough information to solve the first problem in Project Rosalind, Counting DNA Nucleotides. Try the sample problem. If you fixed the sample issue, sign in and try the full version of the problem! Transcribing DNA in RNA . They also have enough information to try out the second problem, transcribing DNA in RNA. Solve the sample problem. Once you have solved the sample problem, log in and try the full version! Complementary to a strand of DNA. You guessed it, you can now also try the third problem: supplementing a strand of DNA. Try the sample problem and then the full version if you are successful. Tuples are basically lists that can never be changed. Lists are very dynamic; They can grow when you attach and paste items, and they can be shrunk when you remove items. You can change any item that you want to change in a list. Sometimes we like this behavior, but sometimes we might want to make sure that no user or part of a program can change a list. That's what the tuples are for. Technically, lists are changeable objects and tuples are immutable objects. Changing objects can change (think mutations), and immutable objects can Define tuples and access to elements.You define a tuple exactly as you define a list, unless you use parentheses instead of square brackets. Once you have a tuple, you can access individual elements, just like you did with a list, and you can loop through the tuple with a for loop: Colors = ('red', 'green', 'blue') print(The first color is: + colors[0]) print(The available colors are:) for colors: print(- + color) The first color is: red The available colors are: - red - green - blue If you try to add something to a tuple, you get an error: colors = 'green', 'blue') colors.append('purple') --- AttributeError Traceback (last call) &lt;ipython-input-37-ed1dbff53ab2&gt;in &lt;module&gt;() 1 Colors = ('red', 'green', 'blue') ----&gt; 2 colors.append('purple'attributes) if you are trying to remove something from a tuple, or change one of its elements. After you define a tuple, you can be sure that its values will not change. We have seen that it is quite useful to mix raw English strings with values stored in variables, such as the following: animal = 'dog' print(I have a + animal + .)) This was especially useful when we had to make a number of similar statements: animals = ['dog', 'cat', 'bear'] for animals in animals: print(I have a + animal + .)) I have a dog. I have a cat. I have a bear. I like this approach of using the plus sign to create strings because it's pretty intuitive. We can see that we are adding several smaller strings to form a longer string. This is intuitive, but it's a lot of typing. There is a shorter way to do this with placeholders. Python ignores most of the characters we put in strings. There are a few characters that Python pays attention to, as we have seen with strings such as -t and. Python also pays attention to %s and %d. These are placeholders. When Python sees the %s placeholder, it looks forward and drags the first argument after the %character: animal = 'dog' print (I have a %s. % animal) This is a much cleaner way to generate strings that contain values. We compose our sentence all in a string and then tell Python which values to drag into the string in the appropriate places. This is called string formatting, and it looks the same when you use a list: Animals = ['Dog', 'Cat', 'Bear'] for animals in animals: print(I have a %s. % animal) I have a dog. I have a cat. I have a bear. If you as a value in the string you compose, you must pack the values into a tuple: animals = ['dog', 'cat', 'bear'] print (I have a %s, a %s and a %s. % (animals[0], animals[1], animals[2])) I have a dog, a cat and a bear. If you remember,&lt;/module&gt; &lt;/ipython-input-37-ed1dbff53ab2&gt; printing a number with a one error: number = 23 print(My favorite number is + number + .) --- TypeError Traceback (last call last) &lt;ipython-input-47-1ed2c5bb2bba&gt;in &lt;module&gt;() 1 number = 23 ----&gt; 2 print(My favorite number is + number + .) TypeError: No str and 'int' objects chain Python knows that you could talk about the value 23 or the characters '23'. So it triggers an error and forces us to clarify that Python should treat the number as a string. We do this by placing the number in a string with the str() function: number = 23 print(My favorite number is + str(number) + .) My favorite number is 23. The format string %d takes care of this for us. See how clean this code is: print number = 23 (My favorite number is %d. % number) My favorite number is 23. If you want to use a series of numbers, pack them into a tuple, as we've seen with strings: numbers = [7, 23, 42] print(My favorite numbers are %d, %d, and %d. % (numbers[0], numbers[1], numbers[2])) My favorite numbers are 7, 23, and 42. For clarity, just look at how long the code is still if you use concatenation instead of string formatting: numbers = [7, 23, 42] print(My favorite numbers are + str(numbers[0]) + , + str(numbers[1]) + and + str(numbers[2]) + .) My favorite numbers are 7, 23 and 42. You can mix string and numeric placeholders in any order. names = ['eric', 'ever'] numbers = [23, 2] print(%s's favorite number is %d, and %s' favorite number is %d. % (names[0].title(), numbers[0], names[1].title(), numbers[1])) Eric's favorite number is 23, and Ever's favorite number is 2. There are more sophisticated ways to do string formatting in Python 3, but we save that for later because it's a bit less intuitive than this approach. At the moment, you can use the approach that gives you consistent output that you want to see. Exercises,Gymnast Scores- A gymnast can earn a score between 1 and 10 from each judge; nothing lower, nothing higher. All values are integer values. There are no decimal values from a single judge. Save the possible points that a gymnast can earn from a judge in a tuple. Print the set The lowest possible score is ___, and the highest possible score is ___. Use the values from your tuple. Print a series of sentences: A judge can give a turner _ points. Don't worry if your first sentence is: a judge can give a gymnast 1 point. However, you will receive 1000 bonus Internet points if you can use a for loop and have the correct grammar. note Revision with Tuples- you have a program that you have already written that uses string concatenation. Save the program with the same file name, but add _tuple.py at the end. For example, gymnast_scores.py becomes gymnast_scores_tuple.py. Instead, rewrite the string sections with %s and %d&lt;/module&gt; &lt;/ipython-input-47-1ed2c5bb2bba&gt; Concatenation. Repeat with two other programs that you have already written. You are now starting to write Python programs that have a little substance. Your programs grow a little longer, and there is a little more structure to your programs. This is a really good time to consider your general style when writing code. Why do we need style conventions? The people who originally developed Python made some of their decisions based on the realization that code is read much more often than it is written. The original developers were equally careful to read the language easily, as well as to write easily. Python has gained a lot of respect as a programming language because the code is readable. You've seen Python use indent to indicate which lines are grouped in a program. This makes the structure of the code visible to anyone who reads it. However, there are some styling decisions that we can make as programmers that can make our programs more readable for ourselves and for others. There are multiple audiences to consider when you think about how readable your code is. You yourself, 6 months from now on. You know what you think when you write code for the first time. But how easy will you remember what you thought when you return to this code tomorrow, next week, or in six months? We want our code to be as easy to read as possible in six months, so that we can get back into our projects if we want. Other programmers you might want to work with. Every major project is the result of cooperation these days. If you stay in programming, you will work with others in jobs and in open source projects. If you write readable code with good commercial letters, people like to work with you in every setting. Potential employers- Most people who hire programmers will ask to see a code you've written, and they'll probably ask you to write code during your interview. If you are used to writing code that is easy to read, you will be good in these situations. What is a PEP?' A PEP is a Python Enhancement Proposal. If users want to suggest changes to the actual Python language, someone designs a Python improvement suggestion. One of the earliest PEPs was a collection of code writing guidelines that is easy to read. It was PEP 8, the style guide for Python code. There's a lot in it that won't make sense to you for some time yet, but there are some suggestions you should know from the beginning. Starting with good Habits now will help you write clean code right from the start that will help you make sense of their code as well. Basic Python Style Policies.Indentation- Use 4 spaces for indentation. This is enough space to give your code a visual structure while leaving room for multiple indentation levels. There are configuration settings in Editors to automatically convert tabs to 4 spaces, and it's a good idea to check this setting. On Geany, this is under Edit&gt;Preferences&gt;Editor&gt;Indentation; set width to 4 and type to spaces. Line Length- Use up to 79 characters per line of code and 72 characters for comments. This is a style policy that some people adhere to and others completely ignore. This refers to a limit on the display size of most monitors. Meanwhile, almost every monitor is able to display much more than 80 characters per line. But we often work in terminals that are not always high-resolution. We also want to have multiple code files open side by side. It turns out that this is still a useful policy that must be followed in most cases. There is a secondary policy on how to hold 99 characters per line if you want longer lines. Many editors have a setting that displays a vertical line that allows you to hold your lines to a certain length. In Geany, you can find this setting under Edit&gt;Preferences&gt;Editor&gt;Display. Make sure Long Line Marker is enabled and set Column to 79. Empty Rows- Use single blank rows to split your code into meaningful blocks. You have seen this in many examples so far. You can use two blank rows in longer programs, but not excessively with blank rows. Use a single space after the pound character at the beginning of a line. If you write more than one paragraph, use a blank line with a pound sign between paragraphs. Name variables and program files that use only lowercase letters, underscores, and numbers. Python won't complain or cause errors if you use capitalization, but you'll mislead other programmers if you use uppercase letters in variables at this point. That's all for the moment. We will go for more style guidelines if we introduce more complicated programming structures. If you follow these guidelines for now, you are well on your way to writing readable code that professionals will respect. If you haven't done so yet, skim PEP 8 - Style Guide for Python Code. As you continue to learn Python, go back and look at this from time to time. I can't stress enough that many good programmers will take you much more seriously from the start if you follow community-wide conventions while writing your code. Implement PEP 8. Take three of your longest programs and add the extension _pep8.py to each program's file name. Revise your code to comply with the styling conventions listed above. Create a list of the most important words you've learned so far in programming. You should have terms such as list, create an appropriate list of definitions. Fill your list with 'Definition'. Use a for loop to print each word and definition. Manage this program until you get to the Python Dictionaries section. These are placed at the bottom, placed, You may have a chance to solve exercises without seeing any clues. Gymnast Scores