26.1 Lists

Survey

The following questions are part of a zyBooks survey to help us improve our content so we can offer the best experience for students. The survey can be taken anonymously and takes just 3-5 minutes. Please take a short moment to answer by clicking the following link.

Link: Student survey

List basics

The ***list*** object type is one of the most important and often used types in a Python program. A list is a ***container***, which is an object that groups related objects together. A list is also a sequence; thus, the contained objects maintain a left-to-right positional ordering. Elements of the list can be accessed via indexing operations that specify the position of the desired element in the list. Each element in a list can be a different type such as strings, integers, floats, or even other lists.

The animation below illustrates how a list is created using brackets `[ ]` around the list elements. The animation also shows how a list object contains references to the contained objects.

PARTICIPATION ACTIVITY 26.1.1: Lists contain references to other objects.

Animation content:

`undefined`

Animation captions:

1. The user creates a new list.
2. The interpreter creates a new object for each list element.
3. 'my_list' holds references to objects in list.

A list can also be created using the built-in list() function. The ***list()*** function accepts a single iterable object argument, such as a string, list, or tuple, and returns a new list object. Ex: list('abc')

creates a new list with the elements ['a', 'b', 'c'].

Accessing list elements

An ***index*** is a zero-based integer matching to a specific position in the list's sequence of elements. A programmer can reference a specific element in a list by providing an index. Ex: `my_list[4]` uses an integer to access the element at index 4 (the 5th element) of my_list.

An index can also be an expression, as long as that expression evaluates into an integer. Replacing the index with an integer variable, such as in `my_list[i]`, allows the programmer to quickly and easily lookup the (i+1)th element in a list.

zyDE 26.1.1: List's ith element can be directly accessed using [i]: Oldest people program.

Consider the following program that allows a user to print the age of the Nth oldest k person to have ever lived. Note: The ages are in a list sorted from oldest to youngest.

1. Modify the program to print the correct ordinal number ("1st", "2nd", "3rd", "4th", "5th of "1th", "2th", "3th", "4th", "5th".
2. For the oldest person, remove the ordinal number (1st) from the print statement to oldest person lived 122 years".

Reminder: List indices begin at 0, not 1, thus the print statement uses oldest_people[nth_person-1], to access the nth_person element (element 1 at index 0 2 at index 1, etc.).

Load default template...

```
1 oldest_people = [122, 119, 117, 117, 116]  #
2
3 nth_person = int(input('Enter N (1-5): '))
4
5 if (nth_person >= 1) and (nth_person <= 5):
6     print(f'The {nth_person}th oldest perso
7
```

3

Run

The program can quickly access the Nth oldest person's age using `oldest_people[nth_person-1]`. Note that the index is `nth_person-1` rather than just nth_person because a list's indices start at 0, so the first age is at index 0, the second at index 1, etc.

A list's index must be an integer type. The index cannot be a floating-point type, even if the value is 0.0, 1.0, etc.

PARTICIPATION ACTIVITY 26.1.2: List indices.

Given the following code, what is the output of each code segment?

```
animals = ['cat', 'dog', 'bird', 'raptor']
print(animals[0])
```

1) `print(animals[0])`

Check Show answer

2) `print(animals[2])`

Check Show answer

3) `print(animals[0 + 1])`

Check Show answer

4)

```
i = 3
print(animals[i])
```


Check Show answer

Modifying a list and common list operations

Unlike the string sequence type, a list is ***mutable*** and is thus able to grow and shrink without the

program having to replace the entire list with an updated copy. Such growing and shrinking capability is called ***in-place modification***. The highlighted lines in the list below indicate ways to perform an in-place modification of the list:

©zyBooks 12/15/22 00:49 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

Table 26.1.1: Some common list operations.

Operation	Description	Example code	Example output
my_list = [1, 2, 3]	Creates a list.	`my_list = [1, 2, 3]` `print(my_list)`	`[1, 2, 3]`
list(iter)	Creates a list.	`my_list = list('123')` `print(my_list)`	`['1', '2', '3']`
my_list[index]	Get an element from a list.	`my_list = [1, 2, 3]` `print(my_list[1])`	`2`
my_list[start:end]	Get a *new* list containing some of another list's elements.	`my_list = [1, 2, 3]` `print(my_list[1:3])`	`[2, 3]`
my_list1 + my_list2	Get a *new* list with elements of my_list2 added to end of my_list1.	`my_list = [1, 2] + [3]` `print(my_list)`	`[1, 2, 3]`

my_list[i] = x	Change the value of the ith element in-place.	`my_list = [1, 2, 3]`
		`my_list[2] = 9`
		`print(my_list)`
			`[1, 2, 9]`

my_list[len(my_list):] = [x]	Add the elements in [x] to the end of my_list. The append(x) method (explained in another section) may be preferred for clarity.	`my_list = [1, 2, 3]`
		`my_list[len(my_list):] = [9]`
		`print(my_list)`
			`[1, 2, 3, 9]`

del my_list[i]	Delete an element from a list.	`my_list = [1, 2, 3]`
		`del my_list[1]`
		`print(my_list)`
			`[1, 3]`

Some of the operations in the table might be familiar to the reader because they are sequence type operations also supported by strings. The dark-shaded rows highlight in-place modification operations.

The below animation illustrates how a program can use in-place modifications to modify the contents of a list.

PARTICIPATION ACTIVITY 26.1.3: In-place modification of a list.

Animation captions:

1. A list, my_list, is created with the chars 'h', 'e', 'l', 'l', and 'o'.
2. Characters ' ', 'w', 'o', 'r', 'l', 'd', and '.' are added to the end of my_list.
3. Index 11 is changed to '!' character.
4. Element at index 5 is removed from my_list.

The difference between in-place modification of a list and an operation that creates an entirely new list is important. In-place modification affects any variable that references the list, and thus can have unintended side-effects. Consider the following code in which the variables your_teams and my_teams reference the same list (via the assignment `your_teams = my_teams`). If either your_teams or my_teams modifies an element of the list, then the change is reflected in the other variable as well.

The below Python Tutor tool executes a Python program and visually shows the objects and variables of a program. The tool shows names of variables on the left, with arrows connecting to bound objects on the right. Note that the tool does not show each number or string character as unique objects to improve clarity. The Python Tutor tool is available at www.pythontutor.com.

PARTICIPATION ACTIVITY 26.1.4: In-place modification of a list.

In the above example, changing the elements of my_teams also affects the contents of your_teams. The change occurs because my_teams and your_teams are bound to the same list object. The code `my_teams[1] = 'Lakers'` modifies the element in position 1 of the shared list object, thus changing the value of both my_teams[1] and your_teams[1].

The programmer of the above example probably meant to only change my_teams. The correct approach would have been to create a *copy* of the list instead. One simple method to create a copy is to use slice notation with no start or end indices, as in `your_teams = my_teams[:]`.

PARTICIPATION ACTIVITY 26.1.5: In-place modification of a copy of a list.

On the other hand, assigning a variable with an element of an existing list, and then reassigning that variable with a different value does not change the list.

PARTICIPATION ACTIVITY 26.1.6: List indexing.

Animation content:

`undefined`

Animation captions:

1. Indexing operation changes list elements.
2. The list does not change when fav_color is updated.

To change the value in the list above, the programmer would have to do an in-place modification operation, such as `colors[1] = 'orange'`.

PARTICIPATION ACTIVITY 26.1.7: List basics.

1) A program can modify the elements of an existing list.

- True
- False

2) The size of a list is determined when the list is created and cannot change.

- True
- False

3) All elements of a list must have the same type.

- True
- False

4) The statement `del my_list[2]` produces a new list without the element in position 2.

- True
- False

5) The statement `my_list3 = my_list1 + my_list2` produces a *new* list.

- True
- False

CHALLENGE ACTIVITY 26.1.1: Enter the output for the list.

422102.2723990.qx3zqy7

Start

Type the program's output

```
user_values = [3, 6, 9]
print(user_values)
```

[3, 6, 9]

1 2 3 4

Check Next

CHALLENGE ACTIVITY 26.1.2: Modify a list.

Modify short_names by deleting the first element and changing the last element to Joe.

Sample output with input: 'Gertrude Sam Ann Joseph'


```
['Sam', 'Ann', 'Joe']
```

422102.2723990.qx3zqy7

```
1 user_input = input()
2 short_names = user_input.split()
3
4 ''' Your solution goes here '''
5
6 print(short_names)
```

Run

26.2 List methods

Common list methods

A ***list method*** can perform a useful operation on a list such as adding or removing elements, sorting, reversing, etc.

The table below shows the available list methods. Many of the methods perform in-place modification of the list contents — a programmer should be aware that a method that modifies the list in-place changes the underlying list object, and thus may affect the value of a variable that

references the object.

©zyBooks 12/15/22 00:49 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

Table 26.2.1: Available list methods.

List method	Description	Code example	Final my_list value
Adding elements			
list.append(x)	Add an item to the end of list.	`my_list = [5, 8]` `my_list.append(16)`	`[5, 8, 16]`
list.extend([x])	Add all items in [x] to list.	`my_list = [5, 8]` `my_list.extend([4, 12])`	`[5, 8, 4, 12]`
list.insert(i, x)	Insert x into list *before* position i.	`my_list = [5, 8]` `my_list.insert(1, 1.7)`	`[5, 1.7, 8]`
Removing elements			
list.remove(x)	Remove first item from list with value x.	`my_list = [5, 8, 14]` `my_list.remove(8)`	`[5, 14]`
list.pop()	Remove and return last item in list.	`my_list = [5, 8, 14]` `val = my_list.pop()`	`[5, 8]` val is 14
list.pop(i)	Remove and return item at position i in list.	`my_list = [5, 8, 14]` `val = my_list.pop(0)`	`[8, 14]` val is 5
Modifying elements			
list.sort()	Sort the items of list in-place.	`my_list = [14, 5, 8]` `my_list.sort()`	`[5, 8, 14]`
	Reverse the elements of list	`my_list = [14, 5, 8]`	`[8, 5`

list.reverse()	in-place.	`my_list.reverse()`	`14]`

Miscellaneous

list.index(x)	Return index of first item in list with value x.	`my_list = [5, 8, 14]` `print(my_list.index(14))`	Prints "2"
list.count(x)	Count the number of times value x is in list.	`my_list = [5, 8, 5, 5, 14]` `print(my_list.count(5))`	Prints "3"

Good practice is to use list methods to add and delete list elements, rather than alternative add/delete approaches. Alternative approaches include syntax such as `my_list[len(my_list):] = [val]` to add to a list, or `del my_list[0]` to remove from a list. Generally, using a list method yields more readable code.

The list.sort() and list.reverse() methods rearrange a list element's ordering, performing in-place modification of the list.

The list.index() and list.count() return information about the list and do not modify the list.

The below interactive tool shows a few of the list methods in action:

PARTICIPATION ACTIVITY 26.2.1: In-place modification using list methods.

Animation content:

`undefined`

Animation captions:

1. vals is a list containing elements 1, 4, and 16.
2. The statement vals.append(9) appends element 9 to the end of the list.
3. The statement vals.insert(2, 18) inserts element 18 into position 2 of the list.
4. The statement vals.pop() removes the last element, 9, from the list.
5. The statement vals.remove(4) removes the first instance of element 4 from the list.
6. The statement vals.remove(55) removes the first instance of element 55 from the list. The list does not contain the element 55 so vals is the same.

zyDE 26.2.1: Amusement park ride reservation system.

The following (unfinished) program implements a digital line queuing system for an amusement park ride. The system allows a rider to reserve a place in line without act having to wait. The rider simply enters a name into a program to reserve a place. Ride purchase a VIP pass get to skip past the common riders up to the last VIP rider in line board the ride first. (Considering the average wait time for a Disneyland ride is about minutes, this might be a useful program.) For this system, an employee manually sel when the ride is dispatched (thus removing the next riders from the front of the line).

Complete the following program, as described above. Once finished, add the followin commands:

- The rider can enter a name to find the current position in line. (Hint: Use the list. method.)
- The rider can enter a name to remove the rider from the line.

Load default ter

```
riders_per_ride = 3  # Num riders per ride to dispatch

line = []  # The line of riders
num_vips = 0  # Track number of VIPs at front of line

menu = ('(1) Reserve place in line.\n'  # Add rider to line
        '(2) Reserve place in VIP line.\n'  # Add VIP
        '(3) Dispatch riders.\n'  # Dispatch next ride car
        '(4) Print riders.\n'
        '(5) Exit.\n\n')

user_input = input(menu).strip().lower()

while user_input != '5':
    if user_input == '1':  # Add rider
        name = input('Enter name:').strip().lower()
        print(name)
```

```
1
Frank
4
```


Run

PARTICIPATION ACTIVITY 26.2.2: List methods.

1) What is the output of the following program?

```
temps = [65, 67, 72, 75]
temps.append(77)
print(temps[-1])
```


Check Show answer

2) What is the output of the following program?

```
actors = ['Pitt',
'Damon']
actors.insert(1,
'Affleck')
print(actors[0],
actors[1], actors[2])
```

Check Show answer

3) Write the simplest two statements that first sort my_list, then remove the largest value element from the list, using list methods.

Check Show answer

4) Write a statement that counts the number of elements of my_list that have the value 15.

Check Show answer

CHALLENGE ACTIVITY 26.2.1: Reverse sort of list.

Sort short_names in reverse alphabetic order.

Sample output with input: 'Jan Sam Ann Joe Tod'

```
['Tod', 'Sam', 'Joe', 'Jan', 'Ann']
```


422102.2723990.qx3zqy7

```
1 user_input = input()
2 short_names = user_input.split()
3
4 ''' Your solution goes here '''
5
6 print(short_names)
```

Run

26.3 Iterating over a list

List iteration

A programmer commonly wants to access each element of a list. Thus, learning how to iterate through a list using a loop is critical.

Looping through a sequence such as a list is so common that Python supports a construct called a ***for loop***, specifically for iteration purposes. The format of a for loop is shown below.

Figure 26.3.1: Iterating through a list.

```
for my_var in my_list:
    # Loop body statements go
here
```


Each iteration of the loop creates a new variable by binding the next element of the list to the name my_var. The loop body statements execute during each iteration and can use the current value of my_var as necessary. [1]

Programs commonly iterate through lists to determine some quantity about the list's items. Ex: The following program determines the value of the maximum even number in a list:

Figure 26.3.2: Iterating through a list example: Finding the maximum even number.

```
user_input = input('Enter numbers:')

tokens = user_input.split()  # Split into separate strings

# Convert strings to integers
nums = []
for token in tokens:
    nums.append(int(token))

# Print each position and number
print()  # Print a single newline
for index in range(len(nums)):
    value = nums[index]

    print(f'{index}: {value}')

# Determine maximum even number
max_num = None
for num in nums:
    if (max_num == None) and (num % 2 == 0):
        # First even number found
        max_num = num
    elif (max_num != None) and (num > max_num ) and (num % 2 == 0):
        # Larger even number found
        max_num = num

print('Max even #:', max_num)
```

```
Enter numbers:3 5 23 -1 456 1 6 83
0: 3
1: 5
2: 23
3: -1
4: 456
5: 1
6: 6
7: 83
Max even #: 456
....
Enter numbers:-5 -10 -44 -2 -27 -9 -27 -9
0:-5
1:-10
2:-44
3:-2
4:-27
5:-9
6:-27
7:-9
Max even #: -2
```


If the user enters the numbers 7, -9, 55, 44, 20, -400, 0, 2, then the program will output

`Max even #: 44`. The code uses three for loops. The first loop converts the strings obtained from the split() function into integers. The second loop prints each of the entered numbers. Note that the first and second loops could easily be combined into a single loop, but the example uses two loops for clarity. The third loop evaluates each of the list elements to find the maximum even number.

Before entering the first loop, the program creates the list nums as an empty list with the statement `nums = []`. The program then appends items to the list inside the first loop. Omitting the initial empty list creation would cause an error when the nums.append() function is called, because nums would not actually exist yet.

The main idea of the code is to use a variable max_num to maintain the largest value seen so far as the program iterates through the list. During each iteration, if the list's current element value is even and larger than max_num so far, then the program assigns max_num with current value. Using a variable to track a value while iterating over a list is very common behavior.

PARTICIPATION ACTIVITY 26.3.1: Using a variable to keep track of a value while iterating over a list.

Animation content:

`undefined`

Animation captions:

1. Loop iterates over all elements of list nums.
2. Only larger even numbers update the value of max_even.
3. Odd numbers, or numbers smaller than max_even, are ignored.
4. When the loop ends, max_even is set to the largest even number 456.

A logic error in the above program would be to set max_even initially to 0, because 0 is not in fact the largest value seen so far. This would result in incorrect output (of 0) if the user entered all negative numbers. Instead, the program sets max_even to None.

PARTICIPATION ACTIVITY 26.3.2: List iteration.

Fill in the missing field to complete the program.

1) Count how many odd numbers (cnt_odd) there are.

```
cnt_odd =
```

```
for i in num:
    if i % 2 == 1:
        cnt_odd += 1
```

2) Count how many negative numbers (cnt_neg) there are.

Check Show answer


```
cnt_neg = 0
for i in num:
    if i < 0:
```

Check Show answer

3) Determine the number of elements in the list that are divisible by 10. (Hint: the number x is divisible by 10 if x % 10 is 0.)

```
div_ten = 0
for i in num:
    if
```

```
        div_ten += 1
```

Check Show answer

IndexError and enumerate()

A common error is to try to access a list with an index that is out of the list's index range, e.g., to try to access my_list[8] when my_list's valid indices are 0-7. Accessing an index that is out of range causes the program to automatically abort execution and generate an ***IndexError***. Ex: For a list my_list containing 8 elements, the statement `my_list[10] = 42` produces output similar to:


```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Iterating through a list for various purposes is an extremely important programming skill to master.

zyDE 26.3.1: Iterating through a list example: Finding the sum of a list's elements.

Here is another example computing the sum of a list of integers. Note that the code i somewhat different than the code computing the max even value. For computing the program initializes a variable sum to 0, then simply adds the current iteration's list ele value to that sum.

Run the program below and observe the output. Next, modify the program to calculat following:

- Compute the average, as well as the sum. Hint: You don't actually have to chan loop, but rather change the printed value.
- Print each number that is greater than 21.

Load default template...

```
# User inputs string w/ numbers: '203 12 5
user_input = input('Enter numbers: ')

tokens = user_input.split()  # Split into

# Convert strings to integers
print()
nums = []
for pos, token in enumerate(tokens):
    nums.append(int(token))
    print(f'{pos}: {token}')

sum = 0
for num in nums:
    sum += num

print('Sum:', sum)
```

```
203 12 5 800 -10
```

Run

The built-in ***enumerate()*** function iterates over a list and provides an iteration counter. The program above uses the enumerate() function, which results in the variables pos and token being assigned the current loop iteration element's index and value, respectively. Thus, the first iteration of the loop assigns pos with 0 and token with the first user number; the second iteration assigns pos with 1 and token with the second user number; and so on.

Built-in functions that iterate over lists

Iterating through a list to find or calculate certain values like the minimum/maximum or sum is so common that Python provides built-in functions as shortcuts. Instead of writing a for loop and tracking a maximum value, or adding a sum, a programmer can use a statement such as

`max(my_list)` or `sum(my_list)` to quickly obtain the desired value.

Table 26.3.1: Built-in functions supporting list objects.

Function	Description	Example code	Example output
all(list)	True if every element in list is True (!= 0), or if the list is empty.	`print(all([1, 2, 3]))` `print(all([0, 1, 2]))`	`True` `False`
any(list)	True if any element in the list is True.	`print(any([0, 2]))` `print(any([0, 0]))`	`True` `False`
max(list)	Get the maximum element in the list.	`print(max([-3, 5, 25]))`	`25`
min(list)	Get the minimum element in the list.	`print(min([-3, 5, 25]))`	`-3`
sum(list)	Get the sum of all elements in the list.	`print(sum([-3, 5, 25]))`	`27`

zyDE 26.3.2: Using built-in functions with lists.

Complete the following program using functions from the table above to find some s about basketball player Lebron James. The code below provides lists of various stati categories for the years 2003-2013. Compute and print the following statistics:

- Total career points
- Average points per game
- Years of the highest and lowest scoring season

Use loops where appropriate.

Load default template... **Run**

```
1
2  #Lebron James: Statistics for 2003/2004 - /
3  games_played = [79, 80, 79, 78, 75, 81, 76
4  points = [1654, 2175, 2478, 2132, 2250, 230
5  assists = [460, 636, 814, 701, 771, 762, 7
6  rebounds = [432, 588, 556, 526, 592, 613, 5
7
8  # Print total points
9
10 # Print Average PPG
11
12 # Print best scoring years (Ex: 2004/2005)
13
14 # Print worst scoring years (Ex: 2004/2005)
15
16
17
```

PARTICIPATION ACTIVITY 26.3.3: Lists and built-in functions.

Assume that my_list is [0, 5, 10, 15].

1) What value is returned by sum(my_list)?

Check **Show answer**

2) What value is returned by max(my_list)?

Check Show answer

3) What value is returned by any(my_list)?

Check Show answer

4) What value is returned by all(my_list)?

Check Show answer

5) What value is returned by min(my_list)?

Check Show answer

CHALLENGE ACTIVITY 26.3.1: Get user guesses.

Write a loop to populate the list user_guesses with a number of guesses. The variable num_guesses is the number of guesses the user will have, which is read first as an integer. Read each guess (an integer) one at a time using int(input()).

Sample output with input:
3
9
5
2


```
user_guesses: [9, 5, 2]
```

422102.2723990.qx3zqy7

```
num_guesses = int(input())
user_guesses = []

''' Your solution goes here '''

print('user_guesses:', user_guesses)
```


Run

CHALLENGE ACTIVITY 26.3.2: Sum extra credit.

Assign sum_extra with the total extra credit received given list test_grades. Iterate through the list with `for grade in test_grades:`. The code uses the Python split() method to split a string at each space into a list of string values and the map() function to convert each string value to an integer. Full credit is 100, so anything over 100 is extra credit.

Sample output for the given program with input: '101 83 107 90'

```
Sum extra: 8
```

(because 1 + 0 + 7 + 0 is 8)

422102.2723990.qx3zqy7

```
user_input = input()
test_grades = list(map(int, user_input.split())) # test_grades is an integer list of test


sum_extra = 0 # Initialize 0 before your loop

''' Your solution goes here '''

print('Sum extra:', sum_extra)
```

Run

CHALLENGE ACTIVITY 26.3.3: Hourly temperature reporting.

Write a loop to print all elements in hourly_temperature. Separate elements with a -> surrounded by spaces.

Sample output for the given program with input: '90 92 94 95'

```
90 -> 92 -> 94 -> 95
```

Note: 95 is followed by a space, then a newline.

422102.2723990.qx3zqy7

```
1 user_input = input()
2 hourly_temperature = user_input.split()
3
4 ''' Your solution goes here '''
5
```

Run

(*1) Actually, a for loop works on any iterable object. An iterable object is any object that can access each of its elements one at a time -- most sequences like lists, strings, and tuples are iterables. Thus, for loops are not specific to lists.

26.4 List games

The following activities can help one become comfortable with iterating through lists. Challenge yourself with these list games.

PARTICIPATION ACTIVITY 26.4.1: Find the maximum value in the list.

If a new maximum value is seen, then click 'Store value'. Try again to get your best time.

Start

max

X	X	X	X	X	X	X

Next value Store value

Time - Best time -

Clear best

PARTICIPATION ACTIVITY 26.4.2: Negative value counting in list.

If a negative value is seen, then click 'Increment'. Try again to get your best time.

Start

Counter

X	X	X	X	X	X	X

0

Next value Increment

Time - Best time -

Clear best

PARTICIPATION ACTIVITY 26.4.3: Manually sorting largest value.

Move the largest value to the right-most position of the list. If the larger of the two current

values is on the left, then swap the values. Try again to get your best time.

Start

X X X X X X X

Next value Swap values

Time - Best time -
Clear best

26.5 List nesting

Since a list can contain any type of object as an element, and a list is itself an object, a list can contain another list as an element. Such embedding of a list inside another list is known as ***list nesting***.Ex: The code `my_list = [[5, 13], [50, 75, 100]]` creates a list with two elements that are each another list.

Figure 26.5.1: Multi-dimensional lists.

```
my_list = [[10, 20], [30, 40]]
print('First nested list:', my_list[0])
print('Second nested list:', my_list[1])
print('Element 0 of first nested list:',
my_list[0][0])
```

```
First nested list: [10, 20]
Second nested list: [30,
40]
Element 0 of first nested
list: 10
```

The program accesses elements of a nested list using syntax such as my_list[0][0].

PARTICIPATION ACTIVITY 26.5.1: List nesting.

Animation captions:

1. The nested lists can be accessed using a single access operation.

2. The elements of each nested list can be accessed using two indexing operations.

PARTICIPATION ACTIVITY 26.5.2: List nesting.

1) Given the list nums = [[10, 20, 30], [98, 99]], what does nums[0][0] evaluate to?

Check Show answer

2) Given the list nums = [[10, 20, 30], [98, 99]], what does nums[1][1] evaluate to?

Check Show answer

3) Given the list nums = [[10, 20, 30], [98, 99]], what does nums[0] evaluate to?

Check Show answer

4) Create a nested list called nums whose only element is the list [21, 22, 23].

Check Show answer

©zyBooks 12/15/22 00:49 1361995 John Farrell COLOSTATECS220SeaboltFall2022

A list is a single-dimensional sequence of items, like a series of times, data samples, daily temperatures, etc. List nesting allows for a programmer to also create a ***multi-dimensional data structure***, the simplest being a two-dimensional table, like a spreadsheet or tic-tac-toe board. The following code defines a two-dimensional table using nested lists:

Figure 26.5.2: Representing a tic-tac-toe board using nested lists.

```
tic_tac_toe = [
    ['X', 'O', 'X'],
    [' ', 'X', ' '],
    ['O', 'O', 'X']
]

print(tic_tac_toe[0][0], tic_tac_toe[0][1], tic_tac_toe[0]
[2])
print(tic_tac_toe[1][0], tic_tac_toe[1][1], tic_tac_toe[1]
[2])
print(tic_tac_toe[2][0], tic_tac_toe[2][1],
tic_tac_toe[2][2])
```

```
X O X
  X
O O X
```


The example above creates a variable tic_tac_toe that represents a 2-dimensional table with 3 rows and 3 columns, for 3*3=9 total table entries. Each row in the table is a nested list. Table entries can be accessed by specifying the desired row and column: tic_tac_toe [1][1] accesses the middle square in row 1, column 1 (starting from 0), which has a value of 'X'. The following animation illustrates:

PARTICIPATION ACTIVITY 26.5.3: Two-dimensional list.

Animation captions:

1. New list object contains other lists as elements.
2. Elements accessed by [row][column].

zyDE 26.5.1: Two-dimensional list example: Driving distance between cities.

The following example illustrates the use of a two-dimensional list in a distance betw example.

Run the following program, entering the text '1 2' as input to find the distance betwee Chicago. Try other pairs. Next, try modifying the program by adding a new city, Ancho is 3400, 3571, and 4551 miles from Los Angeles, Chicago, and Boston, respectively.

Note that the styling of the nested list in this example makes use of indentation to cle indicate the elements of each list -- the spacing does not affect how the interpreter e the list contents.

Load default template...

```
# direct driving distances between cities,
# 0: Boston    1: Chicago    2: Los Angele
distances = [
    [
        0,
        960,  # Boston-Chicago
        2960 # Boston-Los Angeles
    ],
    [
        960,  # Chicago-Boston
        0,
        2011  # Chicago-Los Angeles
    ],
    [
        2960,  # Los Angeles-Boston
        2011,  # Los Angeles-Chicago
```

1 2

Run

The level of nested lists is arbitrary. A programmer might create a three-dimensional list structure as follows:

Figure 26.5.3: The level of nested lists is arbitrary.

```
nested_table = [
    [
        [10, 0,
55],
        [0, 4, 16]
    ],
    [
        [0, 0, 1],
        [1, 20, 2]
    ]
]
```


A number from the above three-dimensional list could be accessed using three indexing operations, as in `nested_table[1][1][1]`.

PARTICIPATION ACTIVITY 26.5.4: Multi-dimensional lists.

Assume the following list has been created

```
scores = [
    [75, 100, 82, 76],
    [85, 98, 89, 99],
    [75, 82, 85, 5]
]
```

1) Write an indexing expression that gets the element from scores whose value is 100.

Check Show answer

2) How many elements does scores contain? (The result of len(scores))

Check Show answer

As always with lists, a typical task is to iterate through the list elements. A programmer can access

all of the elements of nested lists by using ***nested for loops***. The first for loop iterates through the elements of the outer list (rows of a table), while the nested loop iterates through the inner list elements (columns of a table). The code below defines a 3x3 table and iterates through each of the table entries:

PARTICIPATION ACTIVITY 26.5.5: Iterating over multi-dimensional lists.

Animation content:

undefined

Animation captions:

1. Each iteration row is assigned the next list element from currency. Each item in a row is printed in the inner loop.

The outer loop assigns the variable row with one of the list elements. The inner loop then iterates over the elements in that list. Ex: On the first iteration of the outer loop row is [1, 5, 10]. The inner loop then assigns cell 1 on the first iteration, 5 on the second iteration, and 10 on the last iteration.

Combining nested for loops with the enumerate() function gives easy access to the current row and column:

Figure 26.5.4: Iterating through multi-dimensional lists using enumerate().

```
currency = [
   [1, 5, 10 ],  # US Dollars
   [0.75, 3.77, 7.53],  #Euros
   [0.65, 3.25, 6.50]  # British pounds
]
for row_index, row in enumerate(currency):
   for column_index, item in enumerate(row):
       print(f'currency[{row_index}]
[{column_index}] is {item:.2f}')
```

```
currency[0][0] is
1.00
currency[0][1] is
5.00
currency[0][2] is
10.00
currency[1][0] is
0.75
currency[1][1] is
3.77
currency[1][2] is
7.53
currency[2][0] is
0.65
currency[2][1] is
3.25
currency[2][2] is
6.50
```

PARTICIPATION ACTIVITY 26.5.6: Find the error.

The desired output and actual output of each program is given. Find the error in each program.

1) Desired output:

```
0 2 4 6
0 3 6 9 12
```

Actual output:

```
[0, 2, 4, 6] [0, 3, 6, 9, 12]
[0, 2, 4, 6] [0, 3, 6, 9, 12]
```

```
nums = [
   [0, 2, 4, 6],
   [0, 3, 6, 9, 12]
]

for n1
 in nums
:
   for n2
 in nums
:
      print(n2, end=' ')
   print()
```

2) Desired output: `X wins!`

Actual output: `Cat's game!`

```
tictactoe = [
   ['X', 'O', 'O'],
   ['O', 'O', 'X'],
   ['X', 'X', 'X']
]

# Check for 3 Xs in one row
# (Doesn't check columns or diagonals)
for row in tictactoe
:
   num_X_in_row = 0
   for square in row
:
      if square == 'X'
:
         num_X_in_row += 1
   if num_X_in_row == square
:
      print('X wins!')
      break
else:
```

```
print("Cat's game!")
```

CHALLENGE ACTIVITY 26.5.1: Print multiplication table.

Print the two-dimensional list mult_table by row and column. On each line, each character is separated by a space. Hint: Use nested loops.

Sample output with input: '1 2 3,2 4 6,3 6 9':

```
1 | 2 | 3
2 | 4 | 6
3 | 6 | 9
```

422102.2723990.qx3zqy7

```
user_input= input()
lines = user_input.split(',')

# This line uses a construct called a list comprehension, introduced elsewhere,
# to convert the input string into a two-dimensional list.
# Ex: 1 2, 2 4 is converted to [ [1, 2], [2, 4] ]

mult_table = [[int(num) for num in line.split()] for line in lines]

''' Your solution goes here '''

```

Run

26.6 List comprehensions

A programmer commonly wants to modify every element of a list in the same way, such as adding 10 to every element. The Python language provides a convenient construct, known as ***list comprehension***, that iterates over a list, modifies each element, and returns a new list consisting of the modified elements.

A list comprehension construct has the following form:

Construct 26.6.1: List comprehension.

```
new_list = [expression for loop_variable_name in
iterable]
```


A list comprehension has three components:

1. An *expression component* to evaluate for each element in the iterable object.
2. A *loop variable component* to bind to the current iteration element.
3. An *iterable object component* to iterate over (list, string, tuple, enumerate, etc).

A list comprehension is always surrounded by brackets, which is a helpful reminder that the comprehension builds and returns a new list object. The loop variable and iterable object components make up a normal for loop expression. The for loop iterates through the iterable object as normal, and the expression operates on the loop variable in each iteration. The result is a new list containing the values modified by the expression. The below program demonstrates a simple list comprehension that increments each value in a list by 5.

Figure 26.6.1: List comprehension example: A first look.

```
my_list = [10, 20, 30]
list_plus_5 = [(i + 5) for i in
my_list]

print('New list contains:',
list_plus_5)
```

```
New list contains: [15, 25,
35]
```

The following animation illustrates:

PARTICIPATION ACTIVITY 26.6.1: List comprehension.

Animation captions:

1. My list is created and holds integer values.
2. Loop variable i set to each element of my_list.

Programmers commonly prefer using a list comprehension rather than a for loop in many situations. Such preference is due to less code and due to more-efficient execution by the interpreter. The table below shows various for loops and equivalent list comprehensions.

©zyBooks 12/15/22 00:49 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

Table 26.6.1: List comprehensions can replace some for loops.

Num	Description	For loop	Equivalent list comprehension	Output of both pr

1	Add 10 to every element.	`my_list = [5, 20, 50]`
		`for i in range(len(my_list)):`
		`  my_list[ i ] += 10`
		`print(my_list)`
			`my_list = [5, 20, 50]`
			`my_list = [(i+10) for i in my_list]`
			`print(my_list)`
				`[15, 30, 60]`

2	Convert every element to a string.	`my_list = [5, 20, 50]`
		`for i in range(len(my_list)):`
		`  my_list[ i ] = str(my_list[ i ])`
		`print(my_list)`
			`my_list = [5, 20, 50]`
			`my_list = [str(i) for i in my_list]`
			`print(my_list)`
				`['5', '20', '5`

3	Convert user input into a list of integers.	`inp = input('Enter numbers:')`
		`my_list = []`
		`for i in inp.split():`
		
		`my_list.append(int(i))`
		`print(my_list)`
			`inp = input('Enter numbers:')`
			`my_list = [int(i) for i in inp.split()]`
			`print(my_list)`
				`Enter numbers:`
				`[7, 9, 3]`

4	Find the sum of each row in a two-dimensional list.	`my_list = [[5, 10, 15], [2, 3, 16], [100]]`
		`sum_list = []`
		`for row in my_list:`
		
		`sum_list.append(sum(row))`
		`print(sum_list)`
			`my_list = [[5, 10, 15], [2, 3, 16], [100]]`
			`sum_list = [sum(row) for row in my_list]`
			`print(sum_list)`
				`[30, 21, 100]`

5	Find the sum of the row with the smallest sum in a two-dimensional table.	`my_list = [[5, 10, 15], [2, 3, 16], [100]]`
		`sum_list = []`
		`for row in my_list:`
		
		`sum_list.append(sum(row))`
		`min_row = min(sum_list)`
		`print(min_row)`
			`my_list = [[5, 10, 15], [2, 3, 16], [100]]`
			`min_row = min([sum(row) for row in my_list])`
			`print(min_row)`
				`21`

Note that list comprehension is not an exact replacement of for loops, because list comprehensions create a *new* list object, whereas the typical for loop is able to modify an existing list.

The third row of the table above has an expression in place of the iterable object component of the list comprehension, `inp.split()`. That expression is evaluated first, and the list comprehension will loop over the list returned by split().

The last example from above is interesting because the list comprehension is wrapped by the built-in function min(). List comprehension builds a new list when evaluated, so using the new list as an argument to min() is allowed – conceptually the interpreter is just evaluating the more familiar code: min([30, 21, 100]).

PARTICIPATION ACTIVITY 26.6.2: List comprehension examples.

For the following questions, refer to the table above.

1) What's the output of the list comprehension program in row 1 if my_list is [-5, -4, -3]?

Check Show answer

2) Alter the list comprehension from row 2 to convert each number to a float instead of a string.

```
my_list = [5, 20, 50]
my_list = [
                        for
i in my_list]
print(my_list)
```

Check Show answer

3) What's the output of the list comprehension program from row 3 if the user enters "4 6 100"?

Check Show answer

4) What's the output of the list comprehension program in row 4 if my_list is [[5, 10], [1]]?

Check Show answer

5) Alter the list comprehension from row 5 to calculate the sum of every number contained by my_list.

```
 my_list = [[5, 10, 15],
[2, 3, 16], [100]]
sum_list  =
```

```
([sum(row) for row in
my_list])
print(sum_list)
```

Check Show answer

PARTICIPATION ACTIVITY 26.6.3: Building list comprehensions.

Write a list comprehension that contains elements with the desired values. Use the name 'i' as the loop variable.

1) Twice the value of each element in the list variable x.

Check Show answer

2) The absolute value of each element in x. Use the abs() function to find the absolute value of a number.

Check Show answer

3) The largest square root of any element in x. Use math.sqrt() to calculate the square root.

Check Show answer

Conditional list comprehensions

A list comprehension can be extended with an optional conditional clause that causes the statement to return a list with only certain elements.

Construct 26.6.2: Conditional list comprehensions.

```
new_list = [expression for name in iterable if
condition]
```

Using the above syntax will only add an element to the resulting list if the condition evaluates to True. The following program demonstrates list comprehension with a conditional clause that returns a list with only even numbers.

Figure 26.6.2: Conditional list comprehension example: Return a list of even numbers.

```
# Get a list of integers from the user
numbers = [int(i) for i in input('Enter
numbers:').split()]

# Return a list of only even numbers
even_numbers = [i for i in numbers if (i % 2)
== 0]
print('Even numbers only:', even_numbers)
```

```
Enter numbers: 5 52 16 7
25
Even numbers only: [52,
16]
....
Enter numbers: 8 12 -14 9
0
Even numbers only: [8,
12, -14, 0]
```

PARTICIPATION ACTIVITY 26.6.4: Building list comprehensions with conditions.

Write a list comprehension that contains elements with the desired values. Use the name "i" as the loop variable. Use parentheses around the expression or condition as necessary.

1) Only negative values from the list x

```
numbers =
```

Check Show answer

2) Only negative odd values from the list x

```
numbers =
```

Check Show answer

CHALLENGE ACTIVITY 26.6.1: List comprehensions.

422102.2723990.qx3zqy7

Start

Type the program's output

```
my_list = [-1, 0, 1, 2]
new_list = [ number * 3 for number in my_list ]
print(new_list)
```

```
[-3, 0, 3,
```

1 2 3

Check Next

26.7 Sorting lists

One of the most useful list methods is ***sort()***, which performs an in-place rearranging of the list elements, sorting the elements from lowest to highest. The normal relational equality rules are

followed: numbers compare their values, strings compare ASCII/Unicode encoded values, lists compare element-by-element, etc. The following animation illustrates.

PARTICIPATION ACTIVITY 26.7.1: Sorting a list using list.sort().

Animation captions:

1. The list my_list is created and holds integer values.
2. The list is sorted in-place.

The sort() method performs element-by-element comparison to determine the final ordering. Numeric type elements like int and float have their values directly compared to determine relative ordering, i.e., 5 is less than 10.

The below program illustrates the basic usage of the list.sort() method, reading book titles into a list and sorting the list alphabetically.

Figure 26.7.1: list.sort() method example: Alphabetically sorting book titles.

```
books = []
prompt = 'Enter new book: '
user_input = input(prompt).strip()

while (user_input.lower() !=
'exit'):
    books.append(user_input)
    user_input =
input(prompt).strip()

books.sort()

print('\nAlphabetical order:')
for book in books:
    print(book)
```

```
Enter new book: Pride, Prejudice, and
Zombies
Enter new book: Programming in Python
Enter new book: Hackers and Painters
Enter new book: World War Z
Enter new book: exit

Alphabetical order:
Hackers and Painters
Pride, Prejudice, and Zombies
Programming in Python
World War Z
```


The sort() method performs in-place modification of a list. Following execution of the statement my_list.sort(), the contents of my_list are rearranged. The ***sorted()*** built-in function provides the same sorting functionality as the list.sort() method, however, sorted() creates and returns a new list instead of modifying an existing list.

Figure 26.7.2: Using sorted() to create a new sorted list from an existing list without modifying the existing list.

```
numbers = [int(i) for i in input('Enter
numbers: ').split()]

sorted_numbers = sorted(numbers)

print('\nOriginal numbers:', numbers)
print('Sorted numbers:', sorted_numbers)
```

```
Enter numbers: -5 5 -100 23
4 5
Original numbers: [-5, 5,
-100, 23, 4, 5]
Sorted numbers: [-100, -5,
4, 5, 5, 23]
```


PARTICIPATION ACTIVITY 26.7.2: list.sort() and sorted().

1) The sort() method modifies a list in-place.
   - True
   - False

2) The output of the following is [13, 7, 5]:
```
primes = [5, 13, 7]
primes.sort()
print(primes)
```
   - True
   - False

3) The output of `print(sorted([-5, 5, 2]))` is [2, -5, 5].
   - True
   - False

Both the list.sort() method and the built-in sorted() function have an optional ***key*** argument. The key specifies a function to be applied to each element prior to being compared. Examples of key functions are the string methods str.lower, str.upper, or str.capitalize.

Consider the following example, in which a roster of names is sorted alphabetically. If a name is mistakenly uncapitalized, then the sort algorithm places the name at the end of the list, because lower-case letters have a larger encoded value than upper-case letters. Ex: 'a' maps to the ASCII decimal value of 97 and 'A' maps to 65. Specifying the key function as str.lower (note the absence

of parentheses) automatically converts the elements to lower-case before comparison, thus placing the lower-case name at the appropriate position in the sorted list.

Figure 26.7.3: Using the key argument.

```
names = []
prompt = 'Enter name: '

user_input = input(prompt)

while user_input != 'exit':
    names.append(user_input)
    user_input = input(prompt)

no_key_sort = sorted(names)
key_sort = sorted(names, key=str.lower)

print('Sorting without key:', no_key_sort)
print('Sorting with key: ', key_sort)
```



```
Enter name: Serena Williams
Enter name: Venus Williams
Enter name: rafael Nadal
Enter name: john McEnroe
Enter name: exit
Sorting without key: ['Serena Williams', 'Venus Williams', 'john McEnroe', 'rafael
Nadal']
Sorting with key:  ['john McEnroe', 'rafael Nadal', 'Serena Williams', 'Venus
Williams']
```

The key argument can be assigned any function, not just string methods like str.upper and str.lower. Ex: A programmer might want to sort a two-dimensional list by the max of the rows, which can be accomplished by assigning key with the built-in function max, as in: sorted(x, key=max).

©zyBooks 12/15/22 00:49 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

Figure 26.7.4: The key argument to list.sort() or sorted() can be assigned any function.

```
my_list = [[25], [15, 25, 35], [10,
15]]

sorted_list = sorted(my_list,
key=max)

print('Sorted list:', sorted_list)
```

```
Sorted list: [[10, 15], [25], [15,
25, 35]]
```


Sorting also supports the ***reverse*** argument. The reverse argument can be set to a Boolean value, either True or False. Setting reverse=True flips the sorting from lowest-to-highest to highest-to-lowest. Thus, the statement `sorted([15, 20, 25], reverse=True)` produces a list with the elements [25, 20, 15].

PARTICIPATION ACTIVITY 26.7.3: Sorting.

Provide an expression using x.sort that sorts the list x accordingly.

1) Sort the elements of x such that the greatest element is in position 0.

Check Show answer

2) Arrange the elements of x from lowest to highest, comparing the upper-case variant of each element in the list.

Check Show answer

26.8 Command-line arguments

Command-line arguments are values entered by a user when running a program from a command line. A command line exists in some program execution environments, wherein a user can run a program by typing at a command prompt. Ex: To run a Python program named "myprog.py" with an argument specifying the location of a file named "myfile1.txt", the user would enter the following at the command prompt:


```
> python myprog.py myfile1.txt
```

The contents of this command line are automatically stored in the list ***sys.argv***, which is stored in the standard library sys module. sys.argv consists of one string element for each argument typed on the command line.

When executing a program, the interpreter parses the entire command line to find all sequences of characters separated by whitespace, storing each as a string within list variable argv. As the entire command line is passed to the program, the name of the program executable is always added as the first element of the list. Ex: For a command line of `python myprog.py myfile1.txt`, argv has the contents ['myprog.py', 'myfile1.txt'].

The following animation further illustrates.

PARTICIPATION ACTIVITY 26.8.1: Command-line arguments.

Animation captions:

1. Whitespace separates arguments.
2. User text is stored in sys.argv list.

The following program illustrates simple use of command-line arguments, where the program name is myprog, and two additional arguments should be passed to the program.

Figure 26.8.1: Simple use of command line arguments.

```
import sys

name = sys.argv[1]
age = int(sys.argv[2])

print(f'Hello {name}.')
print(f'{age} is a great
age.\n')
```

```
> python myprog.py Tricia 12
Hello Tricia.
12 is a great age.

> python myprog.py Aisha 30
Hello Aisha.
30 is a great age.

> python myprog.py Franco
Traceback (most recent call last):
  File "myprog.py", line 4, in
<module>
    age = sys.argv[2]
IndexError: list index out of range
```


While a program may expect the user to enter certain command-line arguments, there is no guarantee that the user will do so. A common error is to access elements within argv without first checking the length of argv to ensure that the user entered enough arguments, resulting in an IndexError being generated. In the last example above, the user did not enter the age argument, resulting in an IndexError when accessing argv. Conversely, if a user entered too many arguments, extra arguments will be ignored. Above, if the user typed `python myprog.py Alan 70 pizza`, "pizza" will be stored in argv[3] but will never be used by the program.

Thus, when a program uses command-line arguments, a good practice is to always check the length of argv at the beginning of the program to ensure that the user entered the correct number of arguments. The following program uses the statement `if len(sys.argv) != 3` to check for the correct number of arguments, the three arguments being the program, name, and age. If the number of arguments is incorrect, the program prints an error message, referred to as a ***usage message***, that provides the user with an example of the correct command-line argument format. A good practice is to always output a usage message when the user enters incorrect command-line arguments.

Figure 26.8.2: Checking for proper number of command-line arguments.

```
import sys

if len(sys.argv) != 3:
    print('Usage: python myprog.py name age\n')
    sys.exit(1)  # Exit the program, indicating
an error with 1.

name = sys.argv[1]
age = int(sys.argv[2])

print(f'Hello {name}. ')
print(f'{age} is a great age.\n')
```

```
> python myprog.exe
Tricia 12
Hello Tricia. 12 is a
great age.

> python myprog.py
Franco
Usage: python myprog.py
name age

> python myprog.py Alan
70 pizza
Usage: python myprog.py
name age
```


Note that all command-line arguments in argv are strings. If an argument represents a different type like a number, then the argument needs to be converted using one of the built-in functions such as int() or float().

A single command-line argument may need to include a space. Ex: A person's name might be "Mary Jane". Recall that whitespace characters are used to separate the character typed on the command line into the arguments for the program. If the user provided a command line of `python myprog.py Mary Jane 65`, the command-line arguments would consist of four arguments: "myprog.py", "Mary", "Jane", and "65". When a single argument needs to contain a space, the user can enclose the argument within quotes "" on the command line, such as the following, which will result in only 3 command-line arguments, where sys.argv has the contents ['myprog.py', 'Mary Jane', '65'].

```
> python myprog.py "Mary Jane" 65
```

PARTICIPATION ACTIVITY 26.8.2: Command-line arguments.

1) What is the value of sys.argv[1] given the following command-line input (include quotes in your answer): `python prog.py Tricia Miller 26`

Check Show answer

2) What is the value of sys.argv[1] given the following command-line input (include quotes in your answer): `python prog.py 'Tricia Miller' 26`

Check Show answer

Exploring further:

Command-line arguments can become quite complicated for large programs with many options. There are entire modules of the standard library dedicated to aiding a programmer develop sophisticated argument parsing strategies. The reader is encouraged to explore modules such as argparse and getopt.

- argparse: Parser for command-line options, arguments, and sub-commands
- getopt: C-style parser for command-line options

26.9 Additional practice: Engineering examples

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

A list can be useful in solving various engineering problems. One problem is computing the voltage drop across a series of resistors. If the total voltage across the resistors is V, then the current through the resistors will be I = V/R, where R is the sum of the resistances. The voltage drop Vx across resistor x is then Vx = I · Rx.

zyDE 26.9.1: Calculate voltage drops across series of resistors.

The following program uses a list to store a user-entered set of resistance values and computes I.

Modify the program to compute the voltage drop across each resistor, store each in a list voltage_drop, and finally print the results in the following format:


```
5 resistors are in series.
This program calculates the voltage drop across each resistor.
Input voltage applied to circuit: 12.0
Input ohms of 5 resistors
1) 3.3
2) 1.5
3) 2.0
4) 4.0
5) 2.2
Voltage drop per resistor is
1) 3.0 V
2) 1.4 V
3) 1.8 V
4) 3.7 V
5) 2.0 V
```

Load default ter

```
1  num_resistors = 5
2  resistors = []
3  voltage_drop = []
4
5
6  print(f'{num_resistors} resistors are in the series.')
7  print('This program calculates the'),
8  print('voltage drop across each resistor.')
9
10 input_voltage = float(input('Input voltage applied to circuit: '))
11 print (input_voltage)
12
13
14 print(f'Input ohms of {num_resistors} resistors')
15 for i in range(num_resistors):
16
17     res = float(input(f'{i + 1}) '))
```

```
12
3.3
1.5
```


Run

Engineering problems commonly involve matrix representation and manipulation. A matrix can be captured using a two-dimensional list. Then matrix operations can be defined on such lists.

zyDE 26.9.2: Matrix multiplication of 4x2 and 2x3 matrices.

The following illustrates matrix multiplication for 4x2 and 2x3 matrices captured as t
dimensional lists.

Run the program below. Try changing the size and value of the matrices and comput
values.

Load default ter

```
m1_rows = 4
m1_cols = 2
m2_rows = m1_cols  # Must have same value
m2_cols = 3

m1 = [
    [3, 4],
    [2, 3],
    [1, 2],
    [0, 2]
]

m2 = [
    [5, 4, 4],
    [0, 2, 3]
]

```

Run

26.10 Dictionaries

A dictionary is another type of container object that is different from sequences like strings, tuples, and lists. Dictionaries contain references to objects as key-value pairs – each key in the dictionary is associated with a value, much like each word in an English language dictionary is associated with a definition. As of Python 3.7, dictionary elements maintain their insertion order. The ***dict*** type implements a dictionary in Python.

PARTICIPATION ACTIVITY 26.10.1: Dictionaries.

Animation captions:

1. An English dictionary associates words with definitions.
2. A Python dictionary associates keys with values.

There are several approaches to create a dict:

- The first approach wraps braces { } around key-value pairs of literals and/or variables: `{'Jose': 'A+', 'Gino': 'C-'}` creates a dictionary with two keys 'Jose' and 'Gino' that are associated with the grades 'A+' and 'C-', respectively.
- The second approach uses ***dictionary comprehension***, which evaluates a loop to create a new dictionary, similar to how list comprehension creates a new list. Dictionary comprehension is out of scope for this material.
- Other approaches use the ***dict()*** built-in function, using either keyword arguments to specify the key-value pairs or by specifying a list of tuple-pairs. The following creates equivalent dictionaries:
  - dict(Bobby='805-555-2232', Johnny='951-555-0055')
  - dict([('Bobby', '805-555-2232'), ('Johnny', '951-555-0055')])

In practice, a programmer first creates a dictionary and then adds entries, perhaps by reading user-input or text from a file. Dictionaries are mutable, thus entries can be added, modified, or removed in-place. The table below shows some common dict operations.

Table 26.10.1: Common dict operations.

Operation	Description	Example code
my_dict[key]	Indexing operation – retrieves the value associated with key.	`jose_grade = my_dict['Jose']`
my_dict[key] = value	Adds an entry if the entry does not exist, else modifies the existing entry.	`my_dict['Jose'] = 'B+'`
del my_dict[key]	Deletes the key from a dict.	`del my_dict['Jose']`
key in my_dict	Tests for existence of key in my_dict.	`if 'Jose' in my_dict: # ....`

Dictionaries can contain objects of arbitrary type, even other containers such as lists and nested dictionaries. Ex: `my_dict['Jason'] = ['B+', 'A-']` creates an entry in my dict whose value is a list containing the grades of the student 'Jason'.

zyDE 26.10.1: Dictionary example: Gradebook.

Complete the program that implements a gradebook. The student_grades dict should of entries whose keys are student names, and whose values are lists of student score

Load default template...

```
1
2  student_grades = {}  # Create an empty dic
3  grade_prompt = "Enter name and grade (Ex.
4  menu_prompt = ("1. Add/modify student grade
5                 "2. Delete student grade\n"
6                 "3. Print student grades\n"
7                 "4. Quit\n")
8
9  while True:  # Exit when user enters no inp
10     command = input(menu_prompt).lower().st
11     if command == '1':
12         name, grade = input(grade_prompt).s
13         # Your code here
14     elif command == '2':
15         # Your code here
16     elif command == '3':
17         # Your code here
```

Pre-enter any input for program, th run.

Run

PARTICIPATION ACTIVITY 26.10.2: Dictionaries.

1) Dictionary entries can be modified in-place – a new dictionary does not need to be created every time an element is added, changed, or removed.

   - True
   - False

2) The variable my_dict created with the following code contains two keys, 'Bob' and 'A+'.

```
my_dict = dict(name='Bob',
grade='A+')
```

   - True
   - False

CHALLENGE ACTIVITY 26.10.1: Delete from dictionary.

Delete Prussia from country_capital.

Sample output with input: 'Spain:Madrid,Togo:Lome,Prussia: Konigsberg'

```
Prussia deleted? Yes.
Spain deleted? No.
Togo deleted? No.
```


422102.2723990.qx3zqy7

```
user_input = input()
entries = user_input.split(',')
country_capital = {}

for pair in entries:
    split_pair = pair.split(':')
    country_capital[split_pair[0]] = split_pair[1]
    # country_capital is a dictionary, Ex. { 'Germany': 'Berlin', 'France': 'Paris'

''' Your solution goes here '''

print('Prussia deleted?', end=' ')
if 'Prussia' in country_capital:
    print('No.')
else:
    print('Yes.')

```

Run

CHALLENGE ACTIVITY 26.10.2: Enter the output of dictionaries.

422102.2723990.qx3zqy7

Start

Type the program's output

```
airport_codes = {}
airport_codes['Amsterdam'] = 'AMS'
airport_codes['Austin'] = 'AUS'
airport_codes['Tokyo'] = 'NRT'

print(airport_codes['Amsterdam'])
print(airport_codes['Austin'])
```

```
AMS
AUS
```


26.11 Dictionary methods

A ***dictionary method*** is a function provided by the dictionary type (dict) that operates on a specific dictionary object. Dictionary methods can perform some useful operations, such as adding or removing elements, obtaining all the keys or values in the dictionary, merging dictionaries, etc.

Below are a list of common dictionary methods:

Table 26.11.1: Dictionary methods.

Dictionary method	Description	Code example	C

my_dict.clear()	Removes all items from the dictionary.	`my_dict = {'Ahmad': 1, 'Jane': 42}`
		`my_dict.clear()`
		`print(my_dict)`
			`{}`

my_dict.get(key, default)	Reads the value of the key from the dictionary. If the key does not exist in the dictionary, then returns default.	`my_dict = {'Ahmad': 1, 'Jane': 42}`
		`print(my_dict.get('Jane', 'N/A'))`
		`print(my_dict.get('Chad', 'N/A'))`
			`42`
			`N/A`

my_dict1.update(my_dict2)	Merges dictionary my_dict1 with another dictionary my_dict2. Existing entries in my_dict1 are overwritten if the same keys exist in my_dict2.	`my_dict = {'Ahmad': 1, 'Jane': 42}`
		`my_dict.update({'John': 50})`
		`print(my_dict)`
			`{'A`
			`1,`
			`'Ja`
			`42`
			`'Jo`
			`50}`

	Removes and returns the		

my_dict.pop(key, default)	key value from the dictionary. If key does not exist, then default is returned.	`my_dict = {'Ahmad': 1, 'Jane': 42}`
		`val = my_dict.pop('Ahmad')`
		`print(my_dict)`

Modification of dictionary elements using the above methods is performed in-place. Ex: Following the evaluation of the statement my_dict.pop('Ahmad'), any other variables that reference the same object as my_dict will also reflect the removal of 'Ahmad'. As with lists, a programmer should be careful not to modify dictionaries without realizing that other references to the objects may be affected.

PARTICIPATION ACTIVITY 26.11.1: Dictionary methods.

Determine the output of each code segment. If the code produces an error, type None.
Assume that my_dict has the following entries:

```
my_dict = dict(bananas=1.59, fries=2.39, burger=3.50, sandwich=2.99)
```

1)
```
my_dict.update(dict(soda=1.49,
burger=3.69))
burger_price =
my_dict.get('burger', 0)
print(burger_price)
```

Check Show answer

2)
```
my_dict['burger'] =
my_dict['sandwich']
val =
my_dict.pop('sandwich')
print(my_dict['burger'])
```


Check Show answer

CHALLENGE ACTIVITY 26.11.1: Enter the output of dictionary methods.

422102.2723990.qx3zqy7

Start

Type the program's output


```
airport_codes = {
    'New York': 'JFK',
    'Minneapolis': 'MSP',
    'San Francisco': 'SFO',
    'Paris': 'CDG',
    'Austin': 'AUS'
}

print(airport_codes.get('Paris', 'na'))
print(airport_codes.get('Los Angeles', 'na'))
print(airport_codes.get('Seattle', 'na'))
```

```
CDG
na
na
```

1 2 3

Check Next

26.12 Iterating over a dictionary

As usual with containers, a common programming task is to iterate over a dictionary and access or modify the elements of the dictionary. A for loop can be used to iterate over a dictionary object, the loop variable being set to a key of an entry in each iteration. The ordering in which the keys are iterated over is not necessarily the order in which the elements were inserted into the dictionary. The Python interpreter creates a hash of each key. A ***hash*** is a transformation of the key into a unique value that allows the interpreter to perform very fast lookup. Thus, the ordering is actually determined by the hash value, but such hash values can change depending on the Python version and other factors.

Construct 26.12.1: A for loop over a dictionary retrieves each key in the dictionary.

```
for key in dictionary:  # Loop
expression
    # Statements to execute in the
loop

#Statements to execute after the loop
```


The dict type also supports the useful methods items(), keys(), and values() methods, which produce a view object. A ***view object*** provides read-only access to dictionary keys and values. A program can iterate over a view object to access one key-value pair, one key, or one value at a time, depending on the method used. A view object reflects any updates made to a dictionary, even if the dictionary is altered after the view object is created.

- dict.items() – returns a view object that yields (key, value) tuples.
- dict.keys() – returns a view object that yields dictionary keys.
- dict.values() – returns a view object that yields dictionary values.

The following examples show how to iterate over a dictionary using the above methods:

©zyBooks 12/15/22 00:49 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

Figure 26.12.1: Iterating over a dictionary.

dict.items()

```
num_calories = dict(Coke=90,
Coke_zero=0, Pepsi=94)
for soda, calories in
num_calories.items():
    print(f'{soda}:
{calories}')
```

```
Coke: 90
Coke_zero: 0
Pepsi: 94
```


dict.keys()

```
num_calories = dict(Coke=90,
Coke_zero=0, Pepsi=94)
for soda in
num_calories.keys():
    print(soda)
```

```
Coke
Coke_zero
Pepsi
```

dict.values()

```
num_calories = dict(Coke=90,
Coke_zero=0, Pepsi=94)
for soda in
num_calories.values():
    print(soda)
```

```
90
0
94
```

When a program iterates over a view object, one result is generated for each iteration as needed, instead of generating an entire list containing all of the keys or values. Such behavior allows the interpreter to save memory. Since results are generated as needed, view objects do not support indexing. A statement such as my_dict.keys()[0] produces an error. Instead, a valid approach is to use the list() built-in function to convert a view object into a list, and then perform the necessary operations. The example below converts a dictionary view into a list, so that the list can be sorted to find the first two closest planets to Earth.

Figure 26.12.2: Use list() to convert view objects into lists.

```
solar_distances = dict(mars=219.7e6,
venus=116.4e6, jupiter=546e6, pluto=2.95e9)
list_of_distances = list(solar_distances.values())
# Convert view to list

sorted_distance_list = sorted(list_of_distances)
closest = sorted_distance_list[0]
next_closest = sorted_distance_list[1]

print(f'Closest planet is {closest:.4e}')
print(f'Second closest planet is
{next_closest:.4e}')
```

```
Closest planet is
1.1640e+08
Second closest
planet is 2.1970e+08
```


The dict.items() method is particularly useful, as the view object that is returned produces tuples containing the key-value pairs of the dictionary. The key-value pairs can then be unpacked at each iteration, similar to the behavior of enumerate(), providing both the key and the value to the loop body statements without requiring extra code.

zyDE 26.12.1: Iterating over a dictionary example: Gradebook statistics.

Write a program that uses the keys(), values(), and/or items() dict methods to find sta
about the student_grades dictionary. Find the following:

- Print the name and grade percentage of the student with the highest total of pc
- Find the average score of each assignment.
- Find and apply a curve to each student's total score, such that the best student
  100% of the total points.

Load default template... **Run**

```
1
2  # student_grades contains scores (out of 1
3  student_grades = {
4      'Andrew': [56, 79, 90, 22, 50],
5      'Nisreen': [88, 62, 68, 75, 78],
6      'Alan': [95, 88, 92, 85, 85],
7      'Chang': [76, 88, 85, 82, 90],
8      'Tricia': [99, 92, 95, 89, 99]
9  }
10
11
12
```

PARTICIPATION ACTIVITY 26.12.1: Iterating over dictionaries.

Fill in the code, using the dict methods items(), keys(), or values() where appropriate.

1) Print each key in the dictionary my_dict.

```
for key in [          ]:
    print(key)
```


Check Show answer

2) Change all negative values in

my_dict to 0.

```
for key, value in
                  :
    if value < 0:
        my_dict[key] = 0
```

Check Show answer

3) Print twice the value of every value in my_dict.

```
for v in
                  :
    print(2 * v)
```

Check Show answer

CHALLENGE ACTIVITY 26.12.1: Report country population.

Write a loop that prints each country's population in country_pop.

Sample output with input: 'China:1365830000,India:1247220000,United States:318463000,Indonesia:252164800':

```
China has 1365830000 people.
India has 1247220000 people.
United States has 318463000 people.
Indonesia has 252164800 people.
```

422102.2723990.qx3zqy7

```
1  user_input = input()
2  entries = user_input.split(',')
3  country_pop = {}
4
5  for pair in entries:
6      split_pair = pair.split(':')
7      country_pop[split_pair[0]] = split_pair[1]
8      # country_pop is a dictionary, Ex: { 'Germany':'82790000', 'France':'67190000' }
9
10 ''' Your solution goes here '''
11
12     print(f'{country} has {pop} people.')
```

Run

26.13 Dictionary nesting

A dictionary may contain one or more ***nested dictionaries***, in which the dictionary contains another dictionary as a value. Consider the following code:

Figure 26.13.1: Nested dictionaries.

```
students = {}
students ['Jose'] = {'Grade': 'A+', 'StudentID':
22321}

print('Jose:')

print(f' Grade: {students ["Jose"]["Grade"]}')

print(f' ID: {students["Jose"]["StudentID"]}')
```

```
Jose:
 Grade:
A+
 ID:
22321
```

The variable students is first created as an empty dictionary. An indexing operation creates a new entry in students with the key 'Jose' and the value of another dictionary. Indexing operations can be applied to the nested dictionary by using consecutive sets of brackets []: The expression `students['Jose']['Grade']` first obtains the value of the key 'Jose' from students, yielding the nested dictionary. The second set of brackets indexes into the nested dictionary, retrieving the value of the key 'Grade'.

Nested dictionaries also serve as a simple but powerful data structure. A ***data structure*** is a method of organizing data in a logical and coherent fashion. Actually, container objects like lists and dicts are already a form of a data structure, but nesting such containers provides a programmer with much more flexibility in the way that the data can be organized. Consider the simple example below that implements a gradebook using nested dictionaries to organize students and grades.

Figure 26.13.2: Nested dictionaries example: Storing grades.

```
grades = {
    'John Ponting': {
        'Homeworks': [79, 80, 74],
        'Midterm': 85,
        'Final': 92
    },
    'Jacques Kallis': {
        'Homeworks': [90, 92, 65],
        'Midterm': 87,
        'Final': 75
    },
    'Ricky Bobby': {
        'Homeworks': [50, 52, 78],
        'Midterm': 40,
        'Final': 65
    },
}

user_input = input('Enter student name: ')

while user_input != 'exit':
    if user_input in grades:
        # Get values from nested dict
        homeworks = grades[user_input]
['Homeworks']
        midterm = grades[user_input]['Midterm']
        final = grades[user_input]['Final']

        # print info
        for hw, score in enumerate(homeworks):

            print(f'Homework {hw}: {score}')


        print(f'Midterm: {midterm}')

        print(f'Final: {final}')

        # Compute student total score
        total_points = sum([i for i in
homeworks]) + midterm + final

        print(f'Final percentage:
{100*(total_points / 500.0):.1f}%')

    user_input = input('Enter student name: ')
```

```
Enter student name:
Ricky Bobby
Homework 0: 50
Homework 1: 52
Homework 2: 78
Midterm: 40
Final: 65
Final percentage:
57.0%
....
Enter student name:
John Ponting
Homework 0: 79
Homework 1: 80
Homework 2: 74
Midterm: 85
Final: 92
Final percentage:
82.0%
```


©zyBooks 12/15/22 00:49 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

Note the whitespace and indentation used to layout the nested dictionaries. Such layout improves

the readability of the code and makes the hierarchy of the data structure obvious. The extra whitespace does not affect the dict elements, as the interpreter ignores indentation in a multi-line construct.

A benefit of using nested dictionaries is that the code tends to be more readable, especially if the keys are a category like 'Homeworks'. Alternatives like nested lists tend to require more code, consisting of more loops constructs and variables.

Dictionaries support arbitrary levels of nesting; Ex: The expression `students['Jose']['Homeworks'][2]['Grade']` might be applied to a dictionary that has four levels of nesting.

zyDE 26.13.1: Nested dictionaries example: Music library.

The following example demonstrates a program that uses 3 levels of nested dictiona create a simple music library.

The following program uses nested dictionaries to store a small music library. Extenc program such that a user can add artists, albums, and songs to the library. First, add command that adds an artist name to the music dictionary. Then add commands for albums and songs. Take care to check that an artist exists in the dictionary before ac album, and that an album exists before adding a song.

Load default template...

```
music = {
    'Pink Floyd': {
        'The Dark Side of the Moon': {
            'songs': [ 'Speak to Me', 'Brea
            'year': 1973,
            'platinum': True
        },
        'The Wall': {
            'songs': [ 'Another Brick in th
            'year': 1979,
            'platinum': True
        }
    },
    'Justin Bieber': {
        'My World':{
            'songs': ['One Time', 'Bigger'
            'year': 2010,
```

Pre-enter any input for program, th run.

Run

PARTICIPATION ACTIVITY 26.13.1: Nested dictionaries.

1) Nested dictionaries are a flexible way

to organize data.

- True
- False

2) Dictionaries can contain, at most, three levels of nesting.

- True
- False

3) The expression `{'D1': {'D2': 'x'}}` is valid.

- True
- False

26.14 String formatting using %

Conversion specifiers

Program output commonly includes the values of variables as a part of the text. A ***string formatting expression*** allows a programmer to create a string with placeholders that are replaced by the values of variables. Such a placeholder is called a ***conversion specifier***, and different conversion specifiers are used to perform a conversion of the given variable value to a different type when creating the string.

Conversion specifiers convert the object into the desired type. Thus, if the programmer gives a float value of 5.5 to a '%d' conversion specifier, a truncated integer value of '5' is the result.

The syntax for using a conversion specifier also includes a % symbol between the string and the value or variable to be placed into the string. Ex:

```
print('The couch is %d years old.' % couch_age)
```

The '%%' sequence displays an actual percentage sign character, as in

```
print('Annual percentage rate is %f%%' % apr)
```


PARTICIPATION ACTIVITY 26.14.1: Using string formatting expressions.

Animation captions:

1. Simple string replacement can be done using the %s conversion specifier.
2. The %d specifier is used for integer replacement in a formatted string.

3. The %f specifier is used for float replacement in a formatted string. If an integer is passed to a floating-point specifier, the value becomes a float.

Table 26.14.1: Common conversion specifiers.

Conversion specifier(s)	Notes	Example	Output
%d	Substitute as integer.	`print('%d' % 10)`	`10`
%f	Substitute as floating-point decimal	`print('%f' % 15.2)`	`15.200000`
%s	Substitute as string.	`print('%s' % 'ABC')`	`ABC`
%x, %X	Substitute as hexadecimal in lowercase (%x) or uppercase (%X).	`print('%x' % 31)`	`1f`
%e, %E	Substitute as floating-point exponential format in lowercase (%e) or uppercase (%E).	`print('%E' % 15.2)`	`1.520000E+01`

©zyBooks 12/15/22 00:49 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

zyDE 26.14.1: Conversion specifiers automatically convert values.

The program below prints the average payday loan interest rate of 410.9 percent (not see Wikipedia: Payday loan). Try inputting the integer 411 instead, noting how the typ converted by %f to 411.0.

Load default template...

```
1 apr = float(input('Enter APR:\n'))
2
3 # Print using a float conversion specifier
4 print('Annual percentage rate as a float is
5
6 # Print using a decimal conversion specifie
7 print('Annual percentage rate as an int is %
8
```

410.9

Run

PARTICIPATION ACTIVITY 26.14.2: String formatting.

Complete the code using formatting specifiers to generate the described output.

1) Assume `price = 150`.

`I saved $150!`

```
print('I saved $      !'
% price)
```

Check Show answer

2) Assume `percent = 40`.

`Buy now! Save 40%!`

```
print('Buy now! Save
      !' % percent)
```

Check Show answer

Multiple conversion specifiers

Multiple conversion specifiers can appear within the string formatting expression. Expressions that contain more than one conversion specifier must specify the values within a tuple following the '%' character. The following print statement will print a sentence including two numeric values, indicated by the conversion specifiers %d and %f.

Figure 26.14.1: Multiple conversion specifiers.

```
years = 15
total = 500 * (years * 0.02)
print('Savings after %d years is: %f' %
(years, total))
```

```
Savings after 15 years is:
150.000000
```

PARTICIPATION ACTIVITY 26.14.3: Multiple conversion specifiers.

Complete the code using formatting specifiers to generate the described output. Use the indicated variables.

1) Assume `item = 'burrito'` and `price = 5`.

```
The burrito is $5
```

```
print('The      is $%d'
% (item, price))
```

Check Show answer

2) Assume `item = 'backpack'` and `weight = 5.2`.


```
The backpack is 5.200000
pounds.
```

```
print('The %s is
pounds.' % (item, weight))
```

Check Show answer

3) Assume `city = 'Boston'` and `distance = 2100`.

```
We are 2100 miles from
Boston.
```

```
print('We are %d miles
from %s.' % (
                    ))
```

Check Show answer

CHALLENGE ACTIVITY 26.14.1: Printing a string.

Write a *single* statement to print: user_word,user_number. Note that there is no space between the comma and user_number.

Sample output with inputs: 'Amy' 5

```
Amy,5
```

422102.2723990.qx3zqy7

```
1 user_word = str(input())
2 user_number = int(input())
3
4 ''' Your solution goes here '''
5
```


Run

CHALLENGE ACTIVITY 26.14.2: String formatting.

422102.2723990.qx3zqy7

Start

Select the most appropriate conversion specifiers.

num_gifts = 12

print('For Christmas, I got %f gifts from my family.' % num_gifts)

For Christmas, I got 12 gifts from my family.

1 2 3

Check Next

26.15 String formatting using format()

The format() method

Program output commonly includes variables as a part of the text. The string ***format()*** method allows a programmer to create a string with placeholders that are replaced by values or variable values at execution. A placeholder surrounded by curly braces `{ }` is called a ***replacement field***. Values inside the `format()` parentheses are inserted into the replacement fields in the string.

PARTICIPATION ACTIVITY 26.15.1: String formatting.

Animation content:

undefined

Animation captions:

1. The first replacement field {} in the string is replaced with the first value in the format() parentheses.
2. The next replacement field uses the next value, and so on.

The three ways to provide values to replacements fields include:

Table 26.15.1: Three ways to format strings.

Replacement definition	Example	Formatted string result
Positional replacement	`'The {1} in the {0} is {2}.'.format('hat', 'cat', 'fat')`	The cat in the hat is fat.
Inferred positional replacement	`'The {} in the {} is {}.'.format('cat', 'hat', 'fat')`	The cat in the hat is fat.
Named replacement	`'The {animal} in the {headwear} is {shape}.'.format(animal='cat', headwear='hat', shape='fat')`	The cat in the hat is fat.

Named replacement allows a programmer to create a ***keyword argument*** that defines a name and value in the `format()` parentheses. The name can then be placed into a replacement field. Ex: `animal='cat'` is a keyword argument that can be used in a replacement field like `{animal}` to insert the word "cat". Good practice is to use named replacement when formatting strings with many replacement fields to make the code more readable.

Note: The positional and inferred positional replacement types cannot be combined. Ex: `'{} + {1} is {2}'.format(2, 2, 4)` is not allowed. However, named and either positional replacement type can be combined. Ex: `'{} + {} is {sum}'.format(2, 2, sum = 4)`

Double braces `{{ }}` can be used to place an actual curly brace into a string. Ex: `'{0} {{Bezos}}'.format('Amazon')` produces the string "Amazon {Bezos}".

PARTICIPATION ACTIVITY 26.15.2: Positional and named replacement in format strings.

Animation content:

Animation captions:

1. Empty replacement fields infer their position based on the order of values in format().
2. Numbers in replacement fields indicate the position of the value in format().
3. Names in replacement fields indicate a named keyword from format(). Replacement fields can appear more than once in the format.

PARTICIPATION ACTIVITY 26.15.3: string.format() usage.

Determine the output of the following code snippets.

1)
```
print('April {},
{}'.format(22, 2020))
```

Check Show answer

2)
```
date = 'April {}, {}'
print(date.format(22,
2020))
```

Check Show answer

3)
```
date = 'April {}, {}'
print(date.format(22,
2020))
print(date.format(23,
2024))
```

Check Show answer

4)
```
print('{0}:{1}'.format(9,
43))
```

Check Show answer

5)
```
print('{0}:{0}'.format(9,
43))
```


Check Show answer

6)
```
print('Hi
{{{0}}}!'.format('Bilbo'))
```

Check Show answer

7)
```
month = 'April'
day = 22
print('Today is {month}
{0}'.format(day,
month=month))
```

Check Show answer

Format specifications

A ***format specification*** inside of a replacement field allows a value's formatting in the string to be customized. Ex: Using a format specification, a variable with the integer value 4 can be output as a floating-point number (4.0) or with leading zeros (004).

A common format specification is to provide a ***presentation type*** for the value, such as integer (4), floating point (4.0), fixed precision decimal (4.000), percentage (4%), binary (100), etc. A presentation type can be set in a replacement field by inserting a colon `:` and providing one of the presentation type characters described below.

Table 26.15.2: Common formatting specification presentation types.

'

Type	Description	Example	Output
s	String (default presentation type - can be omitted)	`'{:s}'.format('Aiden')`	`Aiden`
d	Decimal (integer values only)	`'{:d}'.format(4)`	`4`
b	Binary (integer values only)	`'{:b}'.format(4)`	`100`
x, X	Hexadecimal in lowercase (x) and uppercase (X) (integer values only)	`'{:x}'.format(15)`	`f`
e	Exponent notation	`'{:e}'.format(44)`	`4.400000e+01`
f	Fixed-point notation (6 places of precision)	`'{:f}'.format(4)`	`4.000000`
.[precision]f	Fixed-point notation (programmer-defined precision)	`'{:.2f}'.format(4)`	`4.00`
0[precision]d	Leading 0 notation	`'{:03d}'.format(4)`	`004`

PARTICIPATION ACTIVITY 26.15.4: Format specifications and presentation types.

Enter the most appropriate format specification to produce the desired output.

1) The value of `num` as a decimal (base 10) integer: `31`

```
num = 31
print('{:
}'.format(num))
```

2) f num as a
Check Show answer
l (base 16) integer:

1f


```
num = 31
print('{:
}'.format(num))
```

Check Show answer

3) The value of num as a binary (base 2) integer: 11111

```
num = 31
print('{:
}'.format(num))
```

Check Show answer

Referencing format() values correctly

The colon : in the replacement field separates the "what" on the left from the "how" on the right. The left "what" side references a value in the format() parentheses. The left side may be omitted (inferred positional replacement), a number (positional replacement), or a name (named replacement). The right "how" side determines how to show the value, such as a presentation type. More advanced format specifications, like fill and alignment, are provided in a later section.

Table 26.15.3: Referencing the correct format() values in replacement fields.

Replacement type	Example	Output
Inferred positional replacement	`'{:s} ${:.2f} tacos is ${:.2f} total'.format('Three', 1.50, 4.50)`	`Three $1.50 tacos is $4.50 total`
Positional replacement	`'{0:s} ${2:.2f} tacos is ${1:.2f} total'.format('Three', 4.50, 1.50)`	`Three $1.50 tacos is $4.50 total`
Named replacement	`'{cnt:s} ${cost:.2f} tacos is ${sum:.2f} total'.format(cnt = 'Three', cost = 1.50, sum = 4.50)`	`Three $1.50 tacos is $4.50 total`

PARTICIPATION ACTIVITY 26.15.5: Matching code blocks to formatted strings.

Match each code block to the code output. If the code would generate an error, mark as "Error".

If unable to drag and drop, refresh the page.

'{} + {} = {}'.format(25, 50, 75)

'{1:} + {0:} = {2:}'.format(25, 50, 75)

'{:.1f} + {:.1f} = {:.2f}'.format(25, 50, 75)

'{} + {} = {2}'.format(25, 50, 75)

'{b} + {a} = {c:.2f}'.format(a=25, b=50, c=75.009)

25 + 50 = 75

50 + 25 = 75

25.0 + 50.0 = 75.00

50 + 25 = 75.01

Error

Reset

CHALLENGE ACTIVITY 26.15.1: String Formatting.

422102.2723990.qx3zqy7

Start

Type the program's output

```
name = 'Joe'

print('Their name is {}'.format(name))
```

Their name is

1 2 3 4 5

Check Next

CHALLENGE ACTIVITY 26.15.2: Printing a string.

Write a *single* statement to print: user_word,user_number. Note that there is no space between the comma and user_number.

Sample output with inputs: Amy 5

```
Amy,5
```


422102.2723990.qx3zqy7

```
1 user_word = input()
2 user_number = int(input())
3
4 ''' Your solution goes here '''
5
```

Run

CHALLENGE ACTIVITY 26.15.3: String formatting.

422102.2723990.qx3zqy7

Start

Select the most appropriate replacement field definitions.

```
num_gifts = 20

print('For Christmas, I got % gifts from my family.'.format(num_gifts))
```

For Christmas, I got 20 gifts from my family.

1 2 3 4

Check Next

26.16 String formatting using dictionaries

Mapping keys

Sometimes a string contains many conversion specifiers. Such strings can be hard to read and understand. Furthermore, the programmer must be careful with the ordering of the tuple values, lest items are mistakenly swapped. A dictionary may be used in place of a tuple on the right side of the conversion operator to enhance clarity at the expense of brevity. If a dictionary is used, then all conversion specifiers must include a ***mapping key*** component. A mapping key is specified by indicating the key of the relevant value in the dictionary within parentheses.

PARTICIPATION ACTIVITY 26.16.1: Using a dictionary and conversion specifiers with mapping keys.

Animation captions:

1. A mapping key is specified by indicating the key of the relevant value in the dict within parentheses.

Figure 26.16.1: Comparing conversion operations using tuples and dicts.

```
import time
gmt = time.gmtime()  # Get current Greenwich Mean Time

print('Time is: {:02d}/{:02d}/{:04d}  {:02d}:{:02d} {:02d} sec' \
     .format(gmt.tm_mon, gmt.tm_mday, gmt.tm_year, gmt.tm_hour,
gmt.tm_min, gmt.tm_sec))
```

```
Time is: 06/07/2013  20:16 24 sec
...
Time is: 06/07/2013  20:16 28 sec
```

```
import time
gmt = time.gmtime()  # Get current Greenwich Mean Time

print('Time is: %(month)02d/%(day)02d/%(year)04d  %(hour)02d:%(min)02d
%(sec)02d sec' %  \
      {
        'year': gmt.tm_year, 'month': gmt.tm_mon, 'day': gmt.tm_mday,
        'hour': gmt.tm_hour, 'min': gmt.tm_min, 'sec': gmt.tm_sec
      }
)
```

```
Time is: 06/07/2013  20:16 24 sec
...
Time is: 06/07/2013  20:16 28 sec
```


PARTICIPATION ACTIVITY 26.16.2: Mapping keys.

Complete the print statement to produce the given output using mapping keys.

1) "I need 12 lilies, 6 roses, and 18 tulips."

```
print ('I need %(lilies)d
lilies, %(roses)d roses,
and %(tulips)d tulips.' %
{
})
```

Check Show answer

2) "My name is Jerome and I'm 15 years old."

```
print ('My name is
%(name)s and I am %(age)d
years old' % {

})
```

Check Show answer

26.17 LAB: Varied amount of input data

Statistics are often calculated with varying amounts of input data. Write a program that takes any number of integers as input, and outputs the average and max.

Ex: If the input is:

```
15 20 0 5
```

the output is:

```
10 20
```

Note: For output, round the average to the nearest integer.

422102.2723990.qx3zqy7

LAB ACTIVITY | 26.17.1: LAB: Varied amount of input data 0 / 10

main.py Load default template...

```
1 ''' Type your code here. '''
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.py** (Your program) → O

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

26.18 LAB: Filter and sort a list

Write a program that gets a list of integers from input, and outputs non-negative integers in ascending order (lowest to highest).

Ex: If the input is:

```
10 -7 4 39 -6 12 2
```


the output is:

```
2 4 10 12 39
```

For coding simplicity, follow every output value by a space. Do not end with newline.

422102.2723990.qx3zqy7

LAB ACTIVITY | 26.18.1: LAB: Filter and sort a list | 0 / 10

main.py Load default template...

```
1 ''' Type your code here. '''
```


Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.py** (Your program) → O

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

26.19 LAB: Middle item

Given a sorted list of integers, output the middle integer. Assume the number of integers is always odd.

Ex: If the input is:


```
2 3 4 8 11
```

the output is:

```
4
```

The maximum number of inputs for any test case should not exceed 9. If exceeded, output "Too many inputs".

Hint: First read the data into a list. Then, based on the list's size, find the middle item.

422102.2723990.qx3zqy7

LAB ACTIVITY | 26.19.1: LAB: Middle item | 0 / 10

main.py — Load default template...

```
1 ''' Type your code here. '''
2
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.py (Your program) → O

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

26.20 LAB: Elements in a range

Write a program that first gets a list of integers from input. That list is followed by two more integers representing lower and upper bounds of a range. Your program should output all integers from the list that are within that range (inclusive of the bounds).

Ex: If the input is:

```
25 51 0 200 33
0 50
```

the output is:

```
25 0 33
```

The bounds are 0-50, so 51 and 200 are out of range and thus not output.

For coding simplicity, follow each output integer by a space, even the last one. Do not end with newline.

422102.2723990.qx3zqy7

LAB ACTIVITY | 26.20.1: LAB: Elements in a range 0 / 10

main.py Load default template...

```
1 ''' Type your code here. '''
```


Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.py** (Your program) → O

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

26.21 LAB: Word frequencies

Write a program that reads a list of words. Then, the program outputs those words and their frequencies (***case insensitive***).

Ex: If the input is:

```
hey Hi Mark hi mark
```


the output is:

```
hey 1
Hi 2
Mark 2
hi 2
mark 2
```

Hint: Use lower() to set each word to lowercase before comparing.

422102.2723990.qx3zqy7

LAB ACTIVITY | 26.21.1: LAB: Word frequencies | 0 / 10

main.py Load default template...

```
1 ''' Type your code here. '''
```


Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.py (Your program) → O

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

26.22 LAB: Contact list

A contact list is a place where you can store a specific contact with other associated information such as a phone number, email address, birthday, etc. Write a program that first takes in word pairs that consist of a name and a phone number (both strings), separated by a comma. That list is followed by a name, and your program should output the phone number associated with that name. Assume the search name is always in the list.

Ex: If the input is:

```
Joe,123-5432 Linda,983-4123 Frank,867-5309
Frank
```

the output is:

```
867-5309
```


422102.2723990.qx3zqy7

LAB ACTIVITY | 26.22.1: LAB: Contact list 0 / 10

main.py Load default template...

```
1 ''' Type your code here. '''
```

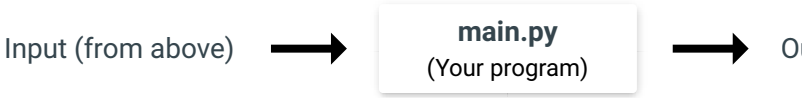
Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program



Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

©zyBooks 12/15/22 00:49 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

26.23 LAB: Replacement words

Write a program that finds word differences between two sentences. The input begins with the first sentence and the following input line is the second sentence. Assume that the two sentences have the same number of words.

The program displays word pairs that differ between the two sentences. One pair is displayed per line.

Ex: If the input is:

```
Smaller cars get better gas mileage
Tiny cars get great fuel economy
```


then the output is:

```
Smaller Tiny
better great
gas fuel
mileage economy
```

Hint: Store each input line into a list of strings.

422102.2723990.qx3zqy7

LAB ACTIVITY | 26.23.1: LAB: Replacement words | 0 / 10

main.py — Load default template...

```
1 ''' Type your code here. '''
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.py (Your program) → O

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

26.24 LAB: Warm up: People's weights (Lists)

(1) Prompt the user to enter four numbers, each corresponding to a person's weight in pounds. Store all weights in a list. Output the list. (2 pts)

Ex:

```
Enter weight 1:
236.0
Enter weight 2:
89.5
Enter weight 3:
176.0
Enter weight 4:
166.3
Weights: [236.0, 89.5, 176.0, 166.3]
```


(2) Output the average of the list's elements with two digits after the decimal point. *Hint: Use a conversion specifier to output with a certain number of digits after the decimal point.* (1 pt)

(3) Output the max list element with two digits after the decimal point. (1 pt)

Ex:

```
Enter weight 1:
```

```
236.0
Enter weight 2:
89.5
Enter weight 3:
176.0
Enter weight 4:
166.3
Weights: [236.0, 89.5, 176.0, 166.3]

Average weight: 166.95
Max weight: 236.00
```


(4) Prompt the user for a number between 1 and 4. Output the weight at the user specified location and the corresponding value in kilograms. 1 kilogram is equal to 2.2 pounds. (3 pts)

Ex:

```
Enter a list location (1 - 4):
3
Weight in pounds: 176.00
Weight in kilograms: 80.00
```

(5) Sort the list's elements from least heavy to heaviest weight. (2 pts)

Ex:

```
Sorted list: [89.5, 166.3, 176.0, 236.0]
```

Output the average and max weights as floating-point values with two digits after the decimal point, which can be achieved as follows:
`print(f'{your_value:.2f}')`

422102.2723990.qx3zqy7

LAB ACTIVITY 26.24.1: LAB: Warm up: People's weights (Lists) 0 / 9

main.py Load default template...

```
1 # FIXME (1): Prompt for four weights. Add all weights to a list. Output list.
2
3 # FIXME (2): Output average of weights.
4
5 # FIXME (3): Output max weight from list.
6
```

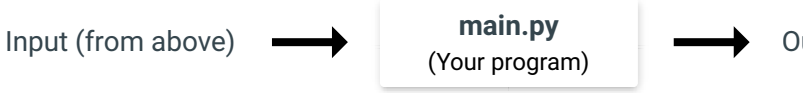
Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program



Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

26.25 LAB*: Program: Soccer team roster (Dictionaries)

This program will store roster and rating information for a soccer team. Coaches rate players during tryouts to ensure a balanced team.

(1) Prompt the user to input five pairs of numbers: A player's jersey number (0 - 99) and the player's

rating (1 - 9). Store the jersey numbers and the ratings in a dictionary. Output the dictionary's elements with the jersey numbers in ascending order (i.e., output the roster from smallest to largest jersey number). *Hint: Dictionary keys can be stored in a sorted list.* (3 pts)

Ex:

```
Enter player 1's jersey number:
84
Enter player 1's rating:
7

Enter player 2's jersey number:
23
Enter player 2's rating:
4

Enter player 3's jersey number:
4
Enter player 3's rating:
5

Enter player 4's jersey number:
30
Enter player 4's rating:
2

Enter player 5's jersey number:
66
Enter player 5's rating:
9

ROSTER
Jersey number: 4, Rating: 5
Jersey number: 23, Rating: 4
Jersey number 30, Rating: 2
...
```


©zyBooks 12/15/22 00:49 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

(2) Implement a menu of options for a user to modify the roster. Each option is represented by a single character. The program initially outputs the menu, and outputs the menu after a user chooses an option. The program ends when the user chooses the option to Quit. For this step, the other options do nothing. (2 pts)

Ex:

```
MENU
a - Add player
d - Remove player
u - Update player rating
r - Output players above a rating
o - Output roster
q - Quit

Choose an option:
```


(3) Implement the "Output roster" menu option. (1 pt)

Ex:

```
ROSTER
Jersey number: 4, Rating: 5
Jersey number: 23, Rating: 4
Jersey number 30, Rating: 2
...
```

(4) Implement the "Add player" menu option. Prompt the user for a new player's jersey number and rating. Append the values to the two vectors. (1 pt)

Ex:

```
Enter a new player's jersey number:
49
Enter the player's rating:
8
```

(5) Implement the "Remove player" menu option. Prompt the user for a player's jersey number. Remove the player from the roster (delete the jersey number and rating). (1 pt)

Ex:

```
Enter a jersey number:
4
```


(6) Implement the "Update player rating" menu option. Prompt the user for a player's jersey number. Prompt again for a new rating for the player, and then change that player's rating. (1 pt)

Ex:

```
Enter a jersey number:
23
Enter a new rating for player:
6
```

(7) Implement the "Output players above a rating" menu option. Prompt the user for a rating. Print the jersey number and rating for all players with ratings above the entered value. (2 pts)

Ex:

```
Enter a rating:
5

ABOVE 5
Jersey number: 66, Rating: 9
Jersey number: 84, Rating: 7
...
```

422102.2723990.qx3zqy7

LAB ACTIVITY 26.25.1: LAB*: Program: Soccer team roster (Dictionaries) 0 / 11

main.py Load default template...

```
1 # Type you code here.
```


Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.py (Your program) → O

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

26.26 LAB: Car wash

This section's content is not available for print.

26.27 LAB: Check if list is sorted

This section's content is not available for print.

26.28 LAB: Scrabble points

This section's content is not available for print.