3.1 String basics

Strings and string literals

A ***string*** is a sequence of characters, like the text MARY, that can be stored in a variable. A ***string literal*** is a string value specified in the source code of a program. A programmer creates a string literal by surrounding text with single or double quotes, such as 'MARY' or "MARY".

The string type is a special construct known as a ***sequence type***: A type that specifies a collection of objects ordered from left to right. A string's characters are ordered from the string's first letter to the last. A character's position in a string is called the character's index, which starts at 0. Ex: In "Trish", T is at index 0, r at 1, etc.

PARTICIPATION ACTIVITY 3.1.1: String indexing.

Type a string below to see how a string is a sequence of characters ordered by position. The numbers on top indicate each character's index.

Type a string
(up to 6 characters)

Trish

0	1	2	3	4	5
T	r	i	s	h	

A programmer can assign a string just as with other types. Ex: str1 = 'Hello', or str1 = str2. The input() function can also be used to get strings from the user.

An empty string is a sequence type with 0 elements, created with two quotes. Ex: `my_str = ''`.

zyDE 3.1.1: A program with strings.

Try the 'mad libs' style game below.

Load default template...

```
brother
burritos
macho
```

Run


```python
#A 'Mad Libs' style game where user enters r
#verbs, etc., and then a story using those w

#Get user's words
relative = input('Enter a type of relative:
print()

food = input('Enter a type of food: ')
print()

adjective = input('Enter an adjective: ')
print()

period = input('Enter a time period: ')
print()

# Tell the story
print('My', relative, 'says eating', food)
print('will make me more', adjective)
print('so now I eat it every', period)
```

PARTICIPATION ACTIVITY

3.1.2: String literals

Indicate which items are string literals.

1) 'Hey'

- Yes
- No

2) 'Hey there.'

- Yes
- No

3) 674

- Yes
- No

4) '674'

- Yes
- No

5) "ok"

- Yes
- No

6) "a"

- Yes
- No

PARTICIPATION ACTIVITY 3.1.3: String basics.

1) Which creates a string variable first_name with a value 'Daniel'?

- `Daniel = first_name`
- `first_name = 'Daniel'`
- `first_name = Daniel`

2) Which prints the value of the first_name variable?

- `print(first_name)`
- `print('first_name')`
- `print("first_name")`

3) Which assigns a string read from input to first_name?

- `first_name = input`
- `input('Type your name:')`
- `first_name = input('Type your name:')`

4) Which answer assigns an empty string to first_name?

- `first_name =`
- `first_name = ''`

○ `'' = first_name`

String length and indexing

A common operation is to find the length, or the number of characters, in a string. The ***len()*** built-in function can be used to find the length of a string (and any other sequence type).

Figure 3.1.1: Using len() to get the length of a string.

The \ character after the string literal extends the string to the following line.

```
george_v = "His Majesty George V, by the Grace
of God, " \
           "of the United Kingdom of Great
Britain and " \
           "Ireland and of the British
Dominions beyond " \
           "the Seas, King, Defender of the
Faith, Emperor of India"
gandhi = 'Mohandas Karamchand Gandhi'
john_f_kennedy = 'JFK'

print(len(george_v), 'characters is much too
long of a name!')
print(len(gandhi), 'characters is better...')
print(len(john_f_kennedy), 'characters is short
enough.')
```

```
185 characters is much too
long of a name!
26 characters is better...
3 characters is short
enough.
```

PARTICIPATION ACTIVITY 3.1.4: Using len() to find the length of a string.

1) What is the length of the string "Santa"?

Check **Show answer**

2) Print the length of the string variable first_name.

Check Show answer

Programs commonly access an individual character of a string. As a sequence type, every character in a string has an index, or position, starting at 0 from the leftmost character. For example, the 'A' in string 'ABC' is at index 0, 'B' is at index 1, and 'C' is at index 2. A programmer can access a character at a specific index by appending ***brackets*** [ ] containing the index:

Figure 3.1.2: Accessing individual characters of a string.

```
alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
print(alphabet[0], alphabet[1], alphabet[25])
```

```
A B Z
```

Note that negative indices can be used to access characters starting from the rightmost character of the string, instead of the leftmost.

zyDE 3.1.2: String indexing.

Try the simple program that looks up the positions of letters in the alpha
negative value like -1, or -25.

Load default template...

2

Run

PARTICIPATION ACTIVITY 3.1.5: String indexing.

1) What character is in index 2 of the string "America"?

Check **Show answer**

2) Write an expression that accesses the first character of the string my_country.

Check **Show answer**

3) Assign my_var with the last character in my_str. Use a negative index.

Check **Show answer**

```
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

user_number = int(input('Enter number to use a
print()

print('\nLetter', user_number, 'of the alphabe
```

Changing string variables and concatenating strings

Writing or altering individual characters of a string variable is not allowed. Strings are immutable objects, meaning that string values cannot change once created. Instead, an assignment statement must be used to update an entire string variable.

Figure 3.1.3: Strings are immutable and cannot be changed.

Individual characters of a string cannot be directly changed.

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'

# Change to upper case

alphabet[0] = 'A'  # Invalid: Cannot
change character
alphabet[1] = 'B'  # Invalid: Cannot
change character

print('Alphabet:', alphabet)
```

```
Traceback (most recent call last):
  File "<stdin>", line 4, in
<module>
TypeError: 'str' object does not
support item assignment
```

Instead, update the variable by assigning an entirely new string.

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'

# Change to upper case
alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

print('Alphabet:', alphabet)
```

```
Alphabet:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```


A program can add new characters to the end of a string in a process known as ***string concatenation***. The expression `"New" + "York"` concatenates the strings New and York to create a new string NewYork. Most sequence types support concatenation.

String concatenation does not contradict the immutability of strings, because the result of concatenation is a new string; the original strings are not altered.

Figure 3.1.4: String concatenation.

```
string_1 = 'abc'
string_2 = '123'
concatenated_string = string_1 + string_2
print('Easy as ' + concatenated_string)
```

```
Easy as abc123
```


PARTICIPATION ACTIVITY 3.1.6: String variables.

1) Python string objects are mutable, meaning that individual characters can be changed.

- True
- False

2) Executing the statements:

```
address = '900 University Ave'
address[0] = '6'
address[1] = '2'
```

is a valid way to change address to '620 University Ave'.

- True
- False

3) Executing the statements:

```
address = '900 University Ave'
address = '620 University Ave'
```

is a valid way to change address to '620 University Ave'.

- True
- False

4)

After the following executes, the value of address is '500 Floral Avenue'.

```
street_num = '500'
street = 'Floral Avenue'
address = street_num + ' ' +
street
```

- True
- False

CHALLENGE ACTIVITY 3.1.1: Reading multiple data types.

Type two statements. The first reads user input into person_name. The second reads user input into person_age. Use the int() function to convert person_age into an integer. Below is a sample output for the given program if the user's input is: Amy 4

```
In 5 years Amy will be 9
```

Note: Do not write a prompt for the input values.

```
person_name = ''
person_age = 0

''' Your solution goes here '''

print('In 5 years', person_name, 'will be', person_age + 5)
```


Run

View solution *(Instructors only)*

Download student submissions

CHALLENGE ACTIVITY 3.1.2: Concatenating strings.

Write two statements to read in values for my_city followed by my_state. Do not provide a prompt. Assign log_entry with current_time, my_city, and my_state. Values should be separated by a space. Sample output for given program if my_city is Houston and my_state is Texas:

```
2014-07-26 02:12:18: Houston Texas
```

Note: Do not write a prompt for the input values.

```
current_time = '2014-07-26 02:12:18:'
my_city = ''
my_state = ''
log_entry = ''

''' Your solution goes here '''

print(log_entry)
```

Run

View solution (Instructors only)

Download student submissions

3.2 List basics

Creating a list

A ***container*** is a construct used to group related values together and contains references to other objects instead of data. A ***list*** is a container created by surrounding a sequence of variables or literals with brackets [ ]. Ex: `my_list = [10, 'abc']` creates a new list variable my_list that contains the two items: 10 and 'abc'. A list item is called an ***element*** .

A list is also a sequence, meaning the contained elements are ordered by position in the list, known as the element's ***index***, starting with 0. `my_list = [ ]`. creates an empty list.

The animation below shows how a list is created and managed by the interpreter. A list itself is an object, and its value is a sequence of references to the list's elements.

PARTICIPATION ACTIVITY 3.2.1: Creating lists.

Animation captions:

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

zyDE 3.2.1: Creating lists.

The following program prints a list of names. Try adding your name to th program again.

Load default template... Run

PARTICIPATION ACTIVITY 3.2.2: Creating lists.

1) Write a statement that creates a list called my_nums, containing the elements 5, 10, and 20.

```
names = ['Daniel', 'Roxanna', 'Jean']

print(names)
```

Check **Show answer**

2) Write a statement that creates a list called my_list with the elements -100 and the string 'lists are fun'.

Check **Show answer**

©zyBooks 03/05/20 10:22 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

3) Write a statement that creates an empty list called class_grades.

Check Show answer

Accessing list elements

Lists are useful for reducing the number of variables in a program. Instead of having a separate variable for the name of every student in a class, or for every word in an email, a single list can store an entire collection of related variables.

Individual list elements can be accessed using an indexing expression by using brackets as in my_list[i], where i is an integer. This allows a programmer to quickly find the i'th element in a list.

Figure 3.2.1: Access list elements using an indexing expression.

```
# Some of the most expensive cars in the
world
lamborghini_veneno = 3900000  # $3.9 million!
bugatti_veyron = 2400000      # $2.4 million!
aston_martin_one77 = 1850000  # $1.85
million!

prices = [lamborghini_veneno, bugatti_veyron,
aston_martin_one77]

print('Lamborghini Veneno:', prices[0],
'dollars')
print('Bugatti Veyron Super Sport:',
prices[1], 'dollars')
print('Aston Martin One-77:', prices[2],
'dollars')
```

```
Lamborghini Veneno: 3900000
dollars
Bugatti Veyron Super Sport:
2400000 dollars
Aston Martin One-77: 1850000
dollars
```


List elements can also be updated with new values by performing an assignment to a position in the list.

A list's index must be an integer. The index cannot be a floating-point type, even if the value is a whole number like 0.0 or 1.0. Using any type besides an integer will produce a runtime error and the program will terminate.

Figure 3.2.2: Modifying list elements

```
my_nums = [5, 12, 20]
print(my_nums)

# Modify a list element
my_nums[1] = -28
print (my_nums)
```

```
[5, 12, 20]
[5, -28, 20]
```


PARTICIPATION ACTIVITY 3.2.3: Accessing list elements.

1) Write a statement that assigns my_var with the 3rd element of my_list.

Check Show answer

2) Write a statement that assigns the 2nd element of my_towns with 'Detroit'.

Check Show answer

CHALLENGE ACTIVITY 3.2.1: Initialize a list.

Initialize the list short_names with strings 'Gus', 'Bob', and 'Zoe'. Sample output for the given program:

```
Gus
Bob
Zoe
```

©zyBooks 03/05/20 10:22 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Run

View solution (Instructors only)

Download student submissions

Adding and removing list elements

Lists are mutable, meaning that a programmer can use methods to add and remove elements from a list as needed. A ***method*** instructs an object to perform some action, and is executed by specifying the method name following a "." symbol and an object. The ***append()*** list method is used to add new elements to a list. Elements can be removed using the ***pop()*** or ***remove()*** methods. Methods are covered in greater detail in another section.

Adding elements to a list:

- list.append(value): Adds value to the end of list. Ex: `my_list.append('abc')`

Removing elements from a list:

- list.pop(i): Removes the element at index i from list. Ex: `my_list.pop(1)`
- list.remove(v): Removes the first element whose value is v. Ex: `my_list.remove('abc')`

PARTICIPATION ACTIVITY 3.2.4: Adding and removing list elements.

Animation content:

```
undefined
```

Animation captions:

1. append() adds an element to the end of the list.
2. pop() removes the element at the given index from the list. 'bw', which is at index 1, is removed and 'abc' is now at index 1.
3. remove() removes the first element with a given value. 'abc' is removed and now the list only has one element.

PARTICIPATION ACTIVITY 3.2.5: List modification.

Write a statement that performs the desired action. Assume the list `house_prices = ['$140,000', '$550,000', '$480,000']` exists.

1) Update the price of the second item in house_prices to '$175,000'.

Check Show answer

2) Add a price to the end of the list with a value of '$1,000,000'.

Check Show answer

3) Remove the 1st element from house_prices, using the pop() method.

Check Show answer

4) Remove '$140,000' from house_prices, using the remove() method.

Check Show answer

Sequence-type methods and functions

Sequence-type functions are built-in functions that operate on sequences like lists and strings. ***Sequence-type methods*** are methods built into the class definitions of sequences like lists and strings. A subset of such functions and methods is provided below.

Table 3.2.1: Some of the functions and methods useful to lists.

Operation	Description
len(list)	Find the length of the list.
list1 + list2	Produce a new list by concatenating list2 to the end of list1.
min(list)	Find the element in list with the smallest value.
max(list)	Find the element in list with the largest value.
sum(list)	Find the sum of all elements of a list (numbers only).
list.index(val)	Find the index of the first element in list whose value matches val.
list.count(val)	Count the number of occurrences of the value val in list.

Figure 3.2.3: Using sequence-type functions with lists.

```
# Concatenating lists
house_prices = [380000, 900000, 875000] + [225000]
print('There are', len(house_prices), 'prices in the
list.')

# Finding min, max
print('Cheapest house:', min(house_prices))
print('Most expensive house:', max(house_prices))
```

```
There are 4 prices in the
list.
Cheapest house: 225000
Most expensive house:
900000
```


Note that lists can contain mixed types of objects. Ex: `x = [1, 2.5, 'abc']` creates a new list x that contains an integer, a floating-point number, and a string. Later material explores lists in detail, including how lists can even contain other lists as elements.

zyDE 3.2.2: Student grade statistics.

The following program calculates some information regarding final and r
enhancing the program by calculating the average midterm and final scc

Lc

```
#Program to calculate statistics from student test scores.
midterm_scores = [99.5, 78.25, 76, 58.5, 100, 87.5, 91, 68, 100]
final_scores = [55, 62, 100, 98.75, 80, 76.5, 85.25]

#Combine the scores into a single list
all_scores = midterm_scores + final_scores

num_midterm_scores = len(midterm_scores)
num_final_scores = len(final_scores)

print(num_midterm_scores, 'students took the midterm.')
print(num_final_scores, 'students took the final.')

#Calculate the number of students that took the midterm but not the fi
dropped_students = num_midterm_scores - num_final_scores
print(dropped_students, 'students must have dropped the class.')

lowest_final = min(final_scores)
highest_final = max(final_scores)

print('\nFinal scores ranged from', lowest_final, 'to', highest_final)
```


Run

PARTICIPATION ACTIVITY 3.2.6: Using sequence-type functions.

1) Write an expression that concatenates the list feb_temps to the end of jan_temps.

Check **Show answer**

2) Write an expression that finds the minimum value in the list prices.

Check **Show answer**

3) Write a statement that assigns the average of the elements of prices to the variable avg_price.

Check **Show answer**

CHALLENGE ACTIVITY 3.2.2: List functions and methods.

Tuples

A ***tuple***, usually pronounced "tuhple" or "toople", behaves similar to a list but is immutable – once created the tuple's elements cannot be changed. A tuple is also a sequence type, supporting len(), indexing, and other sequence type functions. A new tuple is generated by creating a list of comma-separated values, such as `5, 15, 20`.

Typically, tuples are surrounded with parentheses, as in `(5, 15, 20)`. Note that printing a tuple always displays surrounding parentheses.

A tuple is not as common as a list in practical usage, but can be useful when a programmer wants to ensure that values do not change. Tuples are typically used when element position, and not just the relative ordering of elements, is important. Ex: A tuple might store the latitude and longitude of a landmark because a programmer knows that the first element should be the latitude, the second element should be the longitude, and the landmark will never move from those coordinates.

Figure 3.2.4: Using tuples.

```
white_house_coordinates = (38.8977,
77.0366)
print('Coordinates:',
white_house_coordinates)
print('Tuple length:',
len(white_house_coordinates))

# Access tuples via index
print('\nLatitude:',
white_house_coordinates[0], 'north')
print('Longitude:',
white_house_coordinates[1], 'west\n')

# Error. Tuples are immutable
white_house_coordinates[1] = 50
```

```
Coordinates: (38.8977, 77.0366)
Tuple length: 2

Latitude: 38.8977 north
Longitude: 77.0366 west

Traceback (most recent call
last):
  File "<stdin>", line 10, in
<module>
TypeError: 'tuple' object does
not support item assignment
```

PARTICIPATION ACTIVITY 3.2.7: Tuples.

1) Create a new variable `point` that is a tuple containing the strings `'X string'` and `'Y string'`.

Check Show answer

2) If the value of variable `friends` is the tuple `('Cleopatra', 'Marc', 'Seneca')`, then

what is the result of
`len(friends)`?

Check Show answer

CHALLENGE ACTIVITY 3.2.3: Initialize a tuple.

Initialize the tuple `team_names` with the strings 'Rockets', 'Raptors', 'Warriors', and 'Celtics' (The top-4 2018 NBA teams at the end of the regular season in order). Sample output for the given program:

```
Rockets
Raptors
Warriors
Celtics
```

```
team_names = ''' Your solution goes here '''

print(team_names[0])
print(team_names[1])
print(team_names[2])
print(team_names[3])
```


Run

View solution *(Instructors only)*

Download student submissions

Named tuples

A program commonly captures collections of data; for example, a car could be described using a series of variables describing the make, model, retail price, horsepower, and number of seats. A ***named tuple*** allows the programmer to define a new simple data type that consists of named attributes. A `Car` named tuple with fields like `Car.price` and `Car.horsepower` would more clearly represent a car object than a list with index positions correlating to some attributes.

The ***namedtuple*** package must be imported to create a new named tuple. Once the package is imported, the named tuple should be created like in the example below, where the name and attribute names of the named tuple are provided as arguments to the namedtuple constructor. Note that the fields to include in the named tuple are found in a list, but may also be a single string with space or comma separated values.

Figure 3.2.5: Creating named tuples.

```
from collections import namedtuple

Car = namedtuple('Car', ['make','model','price','horsepower','seats'])  #
Create the named tuple

chevy_blazer = Car('Chevrolet', 'Blazer', 32000, 275, 8)  # Use the named
tuple to describe a car
chevy_impala = Car('Chevrolet', 'Impala', 37495, 305, 5)  # Use the named
tuple to describe a different car

print(chevy_blazer)
print(chevy_impala)
```

```
Car(make='Chevrolet', model='Blazer', price=32000, horsepower=275, seats=8)
Car(make='Chevrolet', model='Impala', price=37495, horsepower=305, seats=5)
```


`namedtuple()` only creates the new simple data type, and does not create new data objects. Above, a new data object is not created until `Car()` is called with appropriate values. A data object's attributes can be accessed using dot notation, as in `chevy_blazer.price`. This "named" attribute is simpler to read than if using a list or tuple referenced via index like `chevy_blazer[2]`.

Like normal tuples, named tuples are immutable. A programmer wishing to edit a named tuple would replace the named tuple with a new object.

PARTICIPATION ACTIVITY 3.2.8: Named tuples.

Assume namedtuple has been imported. Use a list of strings in the `namedtuple()` constructor where applicable.

1) Complete the following named tuple definition that describes a house.

```
House =
```

```
('House', ['street',
'postal_code',
'country'])
```

Check **Show answer**

2) Create a new named tuple `Dog` that has the attributes `name`, `breed`, and `color`.

Check **Show answer**

3) Let `Address = namedtuple('Address', ['street', 'city', 'country'])`. Create a new address object `house` where `house.street` is "221B Baker Street", `house.city` is "London", and `country` is "England".

Check **Show answer**

4) Given the following named tuple `Car = namedtuple('Car', ['make', 'model', 'price', 'horsepower', 'seats'])`, and data objects

car1 and car2, write an expression that computes the sum of the price of both cars.

Check Show answer

CHALLENGE ACTIVITY 3.2.4: Creating a named tuple

Define a named tuple `Player` that describes an athlete on a sports team. Include the fields `name`, `number`, `position`, and `team`.

```
from collections import namedtuple

Player = ''' Your solution goes here '''

cam = Player('Cam Newton', '1', 'Quarterback', 'Carolina Panthers')
lebron = Player('Lebron James', '23', 'Small forward', 'Los Angeles Lakers')

print(cam.name + '(#' + cam.number + ')' + ' is a ' + cam.position + ' for the ' +
print(lebron.name + '(#' + lebron.number + ')' + ' is a ' + lebron.position + ' fo
```

Run

View solution *(Instructors only)*

Download student submissions

3.3 Set basics

Set basics

A ***set*** is an unordered collection of unique elements. Sets have the following properties:

- Elements are unordered: Elements in the set do not have a position or index.
- Elements are unique: No elements in the set share the same value.

A set can be created using the ***set()*** function, which accepts a sequence-type iterable object (list, tuple, string, etc.) whose elements are inserted into the set. A ***set literal*** can be written using curly braces `{ }` with commas separating set elements. Note that an empty set can only be created using `set()`.

Figure 3.3.1: Creating sets.

```
# Create a set using the set() function.
nums1 = set([1, 2, 3])

# Create a set using a set literal.
nums2 = { 7, 8, 9 }

# Print the contents of the sets.
print(nums1)
print(nums2)
```

```
{1, 2, 3}
{7, 8, 9}
```

Because the elements of a set are unordered and have no meaningful position in the collection, the index operator is not valid. Attempting to access the element of a set by position, for example `nums1[2]` to access the element at index 2, is invalid and will produce a runtime error.

A set is often used to reduce a list of items that potentially contains duplicates into a collection of unique values. Simply passing a list into `set()` will cause any duplicates to be omitted in the created set.

zyDE 3.3.1: Creating sets.

Load default template...

Run

```
# Initial list contains some duplicate values
first_names = [ 'Harry', 'Hermione', 'Ron', 'H

# Creating a set removes any duplicate values
names_set = set(first_names)

print(names_set)
```


PARTICIPATION ACTIVITY 3.3.1: Basic sets.

1) What's the result of `set(['A', 'Z'])`?

- A set that contains `'A'` and `'Z'`.
- A list with the following elements: `['A', 'Z']`.
- Error: invalid syntax.

2) What's the result of `set(10, 20, 25)`?

- A list with the following elements: `[10, 20, 25]`.
- A set that contains `10`, `20`, and `25`.
- Error: invalid syntax.

©zyBooks 03/05/20 10:22 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

3)

What's the result of `set([100, 200, 100, 200, 300])`?

- A list with the following elements: `[100, 200, 100, 200, 300]`.
- A set that contains `100`, `200`, and `300`.
- A set that contains `100`, `200`, `300`, another `100`, and another `200`.
- Error: invalid syntax.

Modifying sets

Sets are mutable – elements can be added or removed using set methods. The ***add()*** method places a new element into the set if the set does not contain an element with the provided value. The ***remove()*** and ***pop()*** methods remove an element from the set.

Additionally, sets support the `len()` function to return the number of elements in a set. To check if a specific value exists in a set, a membership test such as `value in set` (discussed in another section) can be used.

Adding elements to a set:

- set.add(value): Add value into the set. Ex: `my_set.add('abc')`

Remove elements from a set:

- set.remove(value): Remove the element with given value from the set. Raises KeyError if value is not found. Ex: `my_set.remove('abc')`
- my_set.pop(): Remove a random element from the set. Ex: `my_set.pop()`

Table 3.3.1: Some of the methods useful to sets.

Operation	Description
len(set)	Find the length (number of elements) of the set.
set1.update(set2)	Adds the elements in set2 to set1.

set.add(value)	Adds value into the set.
set.remove(value)	Removes value from the set. Raises KeyError if value is not found.
set.pop()	Removes a random element from the set.
set.clear()	Clears all elements from the set.

PARTICIPATION ACTIVITY 3.3.2: Modifying sets.

Animation content:

Animation captions:

1. Sets can be created using braces {} with commas separating the elements.
2. The add() method adds a single element to a set.
3. The update() method adds the elements of one set to another set.
4. The remove() method removes a single element from a set.
5. The clear() method removes all elements from a set, leaving the set with a length of 0.

PARTICIPATION ACTIVITY 3.3.3: Modifying sets.

Write a line of code to complete the following operations.

1) Add the literal `'Ryder'` to the set `names`.

Check **Show answer**

2) Add all of the elements of set `goblins` into set `monsters`.

3) Remove all of the elements from the `trolls` set.

Check Show answer

4) Get the number of elements in the set `elves`.

Check Show answer

CHALLENGE ACTIVITY 3.3.1: Creating and modifying sets.

The top 3 most popular male names of 2017 are `Oliver`, `Declan`, and `Henry` according to babynames.com.

Write a program that modifies the `male_names` set by removing a name and adding a different name.

Sample output with inputs: 'Oliver' 'Atlas'

```
{ 'Atlas', 'Declan', 'Henry' }
```

NOTE: Because sets are unordered, the order in which the names in `male_names` appear may differ from above.

Run

View solution (Instructors only)

Download student submissions

Set operations

Python set objects support typical set theory operations like intersections and unions. A brief overview of common set operations supported in Python are provided below:

Table 3.3.2: Common set theory operations.

.

Operation	Description
set.intersection(set_a, set_b, set_c...)	Returns a new set containing only the elements in common between `set` and all provided sets.
set.union(set_a, set_b, set_c...)	Returns a new set containing all of the unique elements in all sets.
set.difference(set_a, set_b, set_c...)	Returns a set containing only the elements of `set` that are not found in any of the provided sets.
set_a.symmetric_difference(set_b)	Returns a set containing only elements that appear in exactly one of `set_a` or `set_b`

PARTICIPATION ACTIVITY 3.3.4: Set theory operations.

Animation content:

Animation captions:

1. The union() method builds a set containing the unique elements from names1 and names2. 'Corrin' only appears once in the resulting set.
2. The intersection() method builds a set that contains all common elements between result_set and names3.
3. The difference() method builds a set that contains elements only found in result_set that are not in names4.

PARTICIPATION ACTIVITY 3.3.5: Set theory operations.

Assume that:

- `monsters = {'Gorgon', 'Medusa'}`
- `trolls = {'William', 'Bert', 'Tom'}`
- `horde = {'Gorgon', 'Bert', 'Tom'}`

Fill in the code to complete the line that would produce the given set.

1) `{'Gorgon', 'Bert', 'Tom', 'Medusa', 'William'}`

```
monsters.
(trolls)
```

Check Show answer

2) `{'Gorgon'}`

```
monsters.
(horde)
```

Check Show answer

3) `{'Medusa', 'Bert', 'Tom'}`

```
monsters.symmetric_difference(
                )
```

Check Show answer

CHALLENGE ACTIVITY 3.3.2: Set theory methods.

The following program includes fictional sets of the top 10 male and female baby names for the current year. Write a program that creates:

1. A set `all_names` that contains all of the top 10 male and all of the top 10 female names.
2. A set `neutral_names` that contains only names found in both `male_names` and `female_names`.
3. A set `specific_names` that contains only gender specific names.

Sample output for `all_names`:

```
{'Michael', 'Henry', 'Jayden', 'Bailey', 'Lucas', 'Chuck',
'Aiden', 'Khloe', 'Elizabeth', 'Maria', 'Veronica',
'Meghan', 'John', 'Samuel', 'Britney', 'Charlie', 'Kim'}
```

NOTE: Because sets are unordered, they are printed using the sorted() function here for comparison.

```
male_names = { 'John', 'Bailey', 'Charlie', 'Chuck', 'Michael', 'Samuel', 'Jayd
female_names = { 'Elizabeth', 'Meghan', 'Kim', 'Khloe', 'Bailey', 'Jayden', 'Ai

# Use set methods to create sets all_names, neutral_names, and specific_names.

''' Your solution goes here '''

print(sorted(all_names))
print(sorted(neutral_names))
print(sorted(specific_names))
```

Run

View solution (Instructors only)

Download student submissions

3.4 Dictionary basics

Creating a dictionary

Consider a normal English language dictionary – a reader looks up the word "cat" and finds the definition, "A small, domesticated carnivore." The relationship between "cat" and its definition is *associative*, i.e., "cat" is associated with some words describing "cat."

A ***dictionary*** is a Python container used to describe associative relationships. A dictionary is represented by the ***dict*** object type. A dictionary associates (or "maps") keys with values. A ***key*** is a term that can be located in a dictionary, such as the word "cat" in the English dictionary. A ***value*** describes some data associated with a key, such as a definition. A key can be any immutable type, such as a number, string, or tuple; a value can be any type.

A dict object is created using ***curly braces*** { } to surround the ***key:value pairs*** that comprise the dictionary contents. Ex:
`players = {'Lionel Messi': 10, 'Cristiano Ronaldo': 7}` creates a dictionary called players with two keys: 'Lionel Messi' and 'Cristiano Ronaldo', associated with the values 10 and 7 (their respective jersey numbers). An empty dictionary is created with the expression `players = { }`.

Dictionaries are typically used in place of lists when an associative relationship exists. Ex: If a program contains a collection of anonymous student test scores, those scores should be stored in a list. However, if each score is associated with a student name, a dictionary could be used to associate student names to their score. Other examples of associative relationships include last names and addresses, car models and price, or student ID number and university email address.

Figure 3.4.1: Creating a dictionary.

```
players = {
    'Lionel Messi': 10,
    'Cristiano Ronaldo': 7
}

print(players)
```

```
{'Lionel Messi': 10, 'Cristiano Ronaldo': 7}
```


Note that formatting list or dictionary entries like in the above example, where elements appear on consecutive lines, helps to improve the readability of the code. The behavior of the code is not changed.

zyDE 3.4.1: Creating dictionaries.

Run the program below that displays the caffeine content in milligrams f some popular foods. The indentation and spacing of the caffeine_conter simply provides more readability. Note that order is *not* maintained in the

Try adding new items into the dictionary, using this U.S. federal governm content.

Load default template...

Run

```
caffeine_content_mg = {
    'Mr. Goodbar chocolate': 122,
    'Red Bull': 33,
    'Monster Hitman Sniper energy drink': 270
    'Lipton Brisk iced tea - lemon flavor': 2
    'dark chocolate coated coffee beans': 869
    'Regular drip or percolated coffee': 60,
    'Buzz Bites Chocolate Chews': 1639
}

print(caffeine_content_mg)
```

PARTICIPATION ACTIVITY 3.4.1: Create a dictionary.

1) Use braces to create a dictionary called ages that maps the names 'Bob' and 'Frank' to their ages, 27 and 75, respectively. For this exercise, make 'Bob' the first entry in the dict.

```
ages =
```

Check **Show answer**

Accessing dictionary entries

Unlike a list, which orders elements according to a left-to-right positioning in a sequence, a dictionary's entries maintain no such ordering. To access an entry, the key is specified in brackets [ ]. If no entry with a matching key exists in the dictionary, then a ***KeyError*** runtime error occurs and the program is terminated.

Figure 3.4.2: Accessing dictionary entries.

```
prices = {'apples': 1.99, 'oranges': 1.49}

print('The price of apples is',
prices['apples'])
print('\nThe price of lemons is',
prices['lemons'])
```

```
The price of apples is 1.99
Traceback (most recent call
last):
  File "<stdin>", line 3, in
<module>
KeyError: 'lemons'
```


PARTICIPATION ACTIVITY 3.4.2: Accessing dictionary entries.

1) Dictionary entries are ordered by position.

True

False

2) A dictionary entry is accessed by placing a key in curly braces { }.

True

False

Adding, modifying, and removing dictionary entries

A dictionary is mutable, so entries can be added, modified, and deleted as necessary by a programmer. A new dictionary entry is added by using brackets to specify the key: `prices['banana'] = 1.49`. A dictionary key is unique – attempting to create a new entry with a key that already exists in the dictionary *replaces* the existing entry. The ***del*** keyword is used to remove entries from a dictionary: `del prices['papaya']` removes the entry whose key is 'papaya'. If the requested key to delete does not exist then a KeyError occurs.

Adding new entries to a dictionary:

- dict[k] = v: Adds the new key-value pair k-v, if dict[k] does not already exist.
  Example: `students['John'] = 'A+'`

Modifying existing entries in a dictionary:

- dict[k] = v: Updates the existing entry dict[k], if dict[k] already exists.
  Example: `students['Jessica'] = 'A+'`

Removing entries from a dictionary:

- del dict[k]: Deletes the entry dict[k].
  Example: `del students['Rachel']`

Figure 3.4.3: Adding and editing dictionary entries.

```
{'banana': 1.49}
{'banana': 1.69}
{}
```

```
prices = {}  # Create empty dictionary
prices['banana'] = 1.49  # Add new entry
print(prices)

prices['banana'] = 1.69  # Modify entry
print(prices)

del prices['banana']  # Remove entry
print(prices)
```


PARTICIPATION ACTIVITY 3.4.3: Modifying dictionaries.

1) Which statement adds 'pears' to the following dictionary?

```
prices = {'apples': 1.99,
'oranges': 1.49, 'kiwi': 0.79}
```

- prices['pears'] = 1.79
- prices['pears': 1.79]

2) Executing the following statements produces a KeyError:

```
prices = {'apples': 1.99,
'oranges': 1.49, 'kiwi': 0.79}
del prices['limes']
```

- True
- False

3) Executing the following statements adds a new entry to the dictionary:

```
prices = {'apples': 1.99,
'oranges': 1.49, 'kiwi': 0.79}
prices['oranges'] = 1.29
```

- True
- False

CHALLENGE ACTIVITY 3.4.1: Modify and add to dictionary.

Write a statement to add the key Tesla with value USA to car_makers. Modify the car maker of Fiat to Italy. Sample output for the given program:

```
Acura made in Japan
Fiat made in Italy
Tesla made in USA
```



```python
car_makers = {'Acura': 'Japan', 'Fiat': 'Egypt'}

# Add the key Tesla with value USA to car_makers
# Modify the car maker of Fiat to Italy

''' Your solution goes here '''

print('Acura made in', car_makers['Acura'])
print('Fiat made in', car_makers['Fiat'])
print('Tesla made in', car_makers['Tesla'])
```

Run

View solution (Instructors only)

Download student submissions

3.5 Common data types summary

The most common Python types are presented below.

Common data types

Numeric types int and float represent the most common types used to store data. All numeric types support the normal mathematical operations such as addition, subtraction, multiplication, and division, among others.

Table 3.5.1: Common data types.

Type	Notes
int	Numeric type: Used for variable-width integers.
float	Numeric type: Used for floating-point numbers.

Sequence types string, list, and tuple are all containers for collections of objects ordered by position in the sequence, where the first object has an index of 0 and subsequent elements have indices 1, 2, etc. A list and a tuple are very similar, except that a list is mutable and individual elements may be edited or removed. Conversely, a tuple is immutable and individual elements may not be edited or removed. Lists and tuples can contain any type, whereas a string contains only single-characters. Sequence-type functions such as len() and element indexing using brackets [ ] can be applied to any sequence type.

The only ***mapping type*** in Python is the dict type. Like a sequence type, a dict serves as a container. However, each element of a dict is independent, having no special ordering or relation to other elements. A dictionary uses key-value pairs to associate a key with a value.

Table 3.5.2: Containers: sequence and mapping types.

Type	Notes
string	Sequence type: Used for text.
list	Sequence type: A mutable container with ordered elements.
tuple	Sequence type: An immutable container with ordered elements.
dict	Mapping type: A container with key-values associated elements.

PARTICIPATION ACTIVITY 3.5.1: Common data types.

1) The list `['a', 'b', 3]` is invalid because the list contains a mix of strings and integers.

- True
- False

2) int and float types can always hold the exact same values.

- True
- False

3) A sorted collection of integers might best be contained in a list.

- True
- False

Choosing a container type

New programmers often struggle with choosing the types that best fit their needs, such as choosing whether to store particular data using a list, tuple, or dict. In general, a programmer might use a list when data has an order, such as lines of text on a page. A programmer might use a tuple instead of a list if the contained data should not change. If order is not important, a programmer might use a dictionary to capture relationships between elements, such as student names and grades.

PARTICIPATION ACTIVITY 3.5.2: Choosing among different container types.

Choose the container that best fits the described data.

1) Student test scores that may later be adjusted, ordered from best to worst.

- list

- tuple
- dict

2) Student names and their current grades

- list
- tuple
- dict

3) The final number of As, Bs, Cs, Ds, and Fs in the class

- list
- tuple
- dict

PARTICIPATION ACTIVITY 3.5.3: Finding errors in container code.

Click on the error.

1)

```
# Student grade program.

students = ['Jo', 'Bob', 'Amy']

grades = {}

# Get student name, grade
name = input('name:')
grade = input('grade:')# Assign grade

grades.append(name) = grade
```


2)

```
workers = ('Jo', 'Amy')
```

```
# Remove Amy from workers

del workers[1]

# Print workers

print('Jo:', workers[0])
```


3.6 Additional practice: Grade calculation

The following program calculates an overall grade in a course based on three equally weighted exams.

zyDE 3.6.1: Grade calculator: Average score on three exams.

Load

```
exam1_grade = float(input('Enter score on Exam 1 (out of 100):\n'))
exam2_grade = float(input('Enter score on Exam 2 (out of 100):\n'))
exam3_grade = float(input('Enter score on Exam 3 (out of 100):\n'))

overall_grade = (exam1_grade + exam2_grade + exam3_grade) / 3

print('Your overall grade is:', overall_grade)
```



```
70
75
91
```

Run

Create a different version of the program that:

1. Calculates the overall grade for four equally weighted programming assignments, where each assignment is graded out of 50 points. Hint: First calculate the percentage for each assignment (e.g., score / 50), then calculate the overall grade percentage (be sure to multiply the result by 100).
2. Calculates the overall grade for four equally weighted programming assignments, where assignments 1 and 2 are graded out of 50 points and assignments 3 and 4 are graded out of 75 points.
3. Calculates the overall grade for a course with three equally weighted exams (graded out of 100) that account for 60% of the overall grade and four equally weighted programming assignments (graded out of 50) that account for 40% of the overall graded. Hint: The overall grade can be calculated as 0.6 * averageExamScore + 0.4 * averageProgScore.
4. Extend the program to support the grading scheme for one (or all) of the courses.

3.7 Type conversions

Type conversions

A calculation sometimes must mix integer and floating-point numbers. For example, given that about 50.4% of human births are males, then `0.504 * num_births` calculates the number of expected males in num_births births. If num_births is an integer type, then the expression combines a floating-point and integer.

A ***type conversion*** is a conversion of one type to another, such as an int to a float. An ***implicit conversion*** is a type conversion automatically made by the interpreter, usually

between numeric types. For example, the result of an arithmetic operation like + or * will be a float only if either operand of the operation is a float.

- `1 + 2` returns an integer type.
- `1 + 2.0` returns a float type.
- `1.0 + 2.0` returns a float type.

int-to-float conversion is straightforward: 25 becomes 25.0.

float-to-int conversion just drops the fraction: 4.9 becomes 4.

PARTICIPATION ACTIVITY 3.7.1: Implicit conversions between float and int.

Type the value held in the variable after the assignment statement, given:

- num_items = 5
- item_weight = 0.5

For any floating-point answer, type answer to tenths. Ex: 8.0, 6.5, or 0.1

1) num_items + num_items.

Check Show answer

2) item_weight * num_items.

Check Show answer

3) (num_items + num_items) * item_weight

Check Show answer

Conversion methods

Sometimes a programmer needs to explicitly convert an item's type. Conversion can be explicitly performed using the below conversion methods:

Table 3.7.1: Conversion methods for some common types.

Function	Notes	Can convert:
int()	Creates integers	int, float, strings w/ integers only
float()	Creates floats	int, float, strings w/ integers or fractions
str()	Creates strings	Any

Converting a float to an int will truncate the floating-point number's fraction. For example, the variable temperature might have a value of 18.75232, but can be converted to an integer expression `int(temperature)`. The result would have the value 18, with the fractional part removed.

Conversion of types is very common. In fact, all user input obtained using input() is initially a string and a programmer must explicitly convert the input to a numeric type.

Strings can also be converted to numeric types, if the strings follow the correct formatting, i.e. using only numbers and possibly a decimal point. For example, `int('500')` yields an integer with a value of 500, and `float('1.75')` yields the floating-point value 1.75.

zyDE 3.7.1: Simple example of converting float and int types.

Run the below program. Observe how the type conversion affects the en
the input to 18.552 and run the program again.

Load default template...

18

Run

PARTICIPATION ACTIVITY 3.7.2: Type conversions.

What is the result of each expression?

1) int(1.55)

- 1.55
- 1
- '1.55'

```
input_text = input('Enter a number:\n')
float_variable = float(input_text)
int_variable = int(float_variable)

print('original input text:', input_text)
print('input text converted to a float:', floa
print('float variable converted to an int:', i
```

2) float("7.99")

- 7.0
- 8.0
- 7.99

3) str(99)

- 99
- 99.0
- '99'

CHALLENGE ACTIVITY 3.7.1: Type casting: Computing average owls per zoo.

Assign avg_owls with the average owls per zoo. Print avg_owls as an integer.

Sample output for inputs: 1 2 4

```
Average owls per zoo: 2
```



```
avg_owls = 0.0

num_owls_zooA = int(input())
num_owls_zooB = int(input())
num_owls_zooC = int(input())

''' Your solution goes here '''

print('Average owls per zoo:', int(avg_owls))
```

Run

View solution *(Instructors only)*

Download student submissions

CHALLENGE ACTIVITY 3.7.2: Type casting: Reading and adding values.

Assign total_owls with the sum of num_owls_A and num_owls_B.

Sample output with inputs: 3 4

```
Number of owls: 7
```

```
total_owls = 0

num_owls_A = input()
num_owls_B = input()

''' Your solution goes here '''

print('Number of owls:', total_owls)
```


Run

View solution (Instructors only)

Download student submissions

3.8 Binary numbers

Binary numbers

Normally, a programmer can think in terms of base ten numbers. However, a computer must allocate some finite quantity of bits (e.g., 32 bits) for a variable, and that quantity of bits limits the range of numbers that the variable can represent. Python allocates additional memory to accommodate numbers of very large sizes (past a typical 32 or 64 bit size), and a Python programmer need not think of such low level details. However, binary base computation is a common and important part of computer

science, so some background on how the quantity of bits influences a variable's number range is helpful.

Because each memory location is composed of bits (0s and 1s), a processor stores a number using base 2, known as a ***binary number***.

For a number in the more familiar base 10, known as a ***decimal number***, each digit must be 0-9 and each digit's place is weighed by increasing powers of 10.

Table 3.8.1: Decimal numbers use weighed powers of 10.

Decimal number with 3 digits	Representation
212	$= 2 \cdot 10^2 \quad + 1 \cdot 10^1 \quad + 2 \cdot 10^0$ $= 2 \cdot 100 \quad + 1 \cdot 10 \quad + 2 \cdot 1$ $= 200 \quad + 10 \quad + 2$ $= 212$

In ***base 2***, each digit must be 0-1 and each digit's place is weighed by increasing powers of 2.

Table 3.8.2: Binary numbers use weighed powers of 2.

Binary number with 4 bits	Representation
1101	$= 1 \cdot 2^3 \quad + 1 \cdot 2^2 \quad + 0 \cdot 2^1 \quad + 1 \cdot 2^0$ $= 1 \cdot 8 \quad + 1 \cdot 4 \quad + 0 \cdot 2 \quad + 1 \cdot 1$ $= 8 \quad + 4 \quad + 0 \quad + 1$ $= 13$

PARTICIPATION

ACTIVITY 3.8.1: Binary number tool.

Set each binary digit for the unsigned binary number below to 1 or 0 to obtain the decimal equivalents of 9, then 50, then 212, then 255. Note also that 255 is the largest integer that the 8 bits can represent.

PARTICIPATION ACTIVITY 3.8.2: Binary numbers.

1) Convert the binary number 00001111 to a decimal number.

Check **Show answer**

2) Convert the binary number 10001000 to a decimal number.

Check **Show answer**

3) Convert the decimal number 17 to an 8-bit binary number.

Check **Show answer**

4) Convert the decimal number 51 to an 8-bit binary number.

Check **Show answer**

©zyBooks 03/05/20 10:22 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

CHALLENGE ACTIVITY 3.8.1: Create a binary number.

3.9 String formatting

The format() function

Program output commonly includes variables as a part of the text. The string ***format()*** function 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 3.9.1: 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 3.9.1: 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 3.9.2: 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().

PARTICIPATION ACTIVITY 3.9.3: 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 3.9.2: 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		`4.400000e+01`

		`'{:e}'.format(44)`	
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`

PARTICIPATION ACTIVITY 3.9.4: 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))
```

Check **Show answer**

2) The value of `num` as a hexadecimal (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 3.9.3: 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 3.9.5: Matching code blocks to formatted strings.

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

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

25 + 50 = 75

50 + 25 = 75

25.0 + 50.0 = 75.00

50 + 25 = 75.01

Error

Reset

CHALLENGE ACTIVITY 3.9.1: 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
```

```
user_word = str(input())
user_number = int(input())

''' Your solution goes here '''

```

Run

View solution (Instructors only)

Download student submissions

CHALLENGE ACTIVITY 3.9.2: String formatting.

3.10 Additional practice: Health data

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.

The following calculates a user's age in days based on the user's age in years.

zyDE 3.10.1: Health data: Age in days.

Load default template...

22

Run

Create a different version of the program that:

1. Calculates the user's age in minutes and seconds.
2. Estimates the approximate number of times the user's heart has beat in his/her lifetime using an average heart rate of 72 beats per minute.
3. Estimates the number of times the person has sneezed in his/her lifetime.
4. Estimates the number of calories that the person has expended in his/her lifetime (research on the Internet to obtain a daily estimate). Also calculate the number of sandwiches (or other common food item) that equals that number of calories.
5. Be creative: Pick several other interesting health-related statistics. Try searching the Internet to determine how to calculate that data, and create a program to perform that calculation. The program can ask the user to enter any information needed to perform the calculation.

3.11 LAB: Input and formatted output: Right-facing arrow

Given input characters for an arrowhead and arrow body, print a right-facing arrow.

Ex: If the input is:

```
*

#
```

```
user_age_years = int(input('enter your age in

user_age_days = user_age_years  * 365

print('You are at least {} days old.'.format(u

```

Then the output is:


```
       #
******##
******###
******##
       #
```

LAB ACTIVITY 3.11.1: LAB: Input and formatted output: Right-facing arrow 0 / 10

main.py

Load default template...

```
base_char = input()
head_char = input()

row1 = '      ' + head_char
''' Type your code here. '''

print(row1)
print(row2)
print(row3)
print(row2)
print(row1)

```

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) → Output (shown be

Program output displayed here

Lab statistics and submissions **Show**

Solution **Show**

Tests **Show**

3.12 LAB: Phone number breakdown

Given an integer representing a 10-digit phone number, output the area code, prefix, and line number using the format (800) 555-1212.

Ex: If the input is:


```
8005551212
```

the output is:

```
(800) 555-1212
```

Hint: Use % to get the desired rightmost digits. Ex: The rightmost 2 digits of 572 is gotten by 572 % 100, which is 72.

Hint: Use // to shift right by the desired amount. Ex: Shifting 572 right by 2 digits is done by 572 // 100, which yields 5. (Recall integer division discards the fraction).

For simplicity, assume any part starts with a non-zero digit. So 0119998888 is not allowed.

LAB ACTIVITY 3.12.1: LAB: Phone number breakdown 0 / 10

main.py

Load default template...

```
phone_number = int(input())

''' 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) → Output (shown be

Program output displayed here

Lab statistics and submissions Show

Solution Show

Tests Show

3.13 LAB: Input and formatted output: Caffeine levels

A half-life is the amount of time it takes for a substance or entity to fall to half its original value. Caffeine has a half-life of about 6 hours in humans. Given caffeine amount (in mg) as input, output the caffeine level after 6, 12, and 24 hours. Use a string formatting expression with conversion specifiers to output the caffeine amount as floating-point numbers.

Output each floating-point value with two digits after the decimal point, which can be achieved as follows:

`print('{:.2f}'.format(your_value))`

Ex: If the input is:

```
100
```

the output is:

```
After 6 hours: 50.00 mg
After 12 hours: 25.00 mg
After 24 hours: 6.25 mg
```

Note: A cup of coffee has about 100 mg. A soda has about 40 mg. An "energy" drink (a misnomer) has between 100 mg and 200 mg.

LAB ACTIVITY 3.13.1: LAB: Input and formatted output: Caffeine levels 0 / 10

main.py

Load default template...

```
caffeine_mg = float(input())

''' 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) → Output (shown be

Program output displayed here

Lab statistics and submissions Show

Solution Show

Tests Show

3.14 LAB: Input and formatted output: House real estate summary

Sites like Zillow get input about house prices from a database and provide nice summaries for readers. Write a program with two inputs, current price and last month's price (both integers). Then, output a summary listing the price, the change since last month, and the estimated monthly mortgage computed as (current_price * 0.051) / 12.

Output each floating-point value with two digits after the decimal point, which can be achieved as follows:
`print('{:.2f}'.format(your_value))`

Ex: If the input is:


```
200000
210000
```

the output is:

```
This house is $200000. The change is $-10000 since last
month.
The estimated monthly mortgage is $850.00.
```

Note: Getting the precise spacing, punctuation, and newlines *exactly* right is a key point of this assignment. Such precision is an important part of programming.

LAB ACTIVITY 3.14.1: LAB: Input and formatted output: House real estate summary

0 / 10

main.py

Load default template...

```
current_price = int(input())
last_months_price = int(input())

''' 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) → Output (shown be

Program output displayed here

Lab statistics and submissions Show

Solution Show

Tests Show

3.15 LAB: Simple statistics

Given 4 floating-point numbers. Use a string formatting expression with conversion specifiers to output their product and their average as integers (rounded), then as floating-point numbers.

Output each rounded integer using the following:
`print('{:.0f}'.format(your_value))`

Output each floating-point value with three digits after the decimal point, which can be achieved as follows:
`print('{:.3f}'.format(your_value))`

Ex: If the input is:


```
8.3
10.4
5.0
4.8
```

the output is:

```
2072 7
2071.680 7.125
```

LAB ACTIVITY 3.15.1: LAB: Simple statistics 0 / 10

main.py

Load default template...

```
num1 = float(input())
num2 = float(input())
num3 = float(input())

''' 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) → Output (shown be

Program output displayed here

Lab statistics and submissions Show

Solution Show

Tests Show

3.16 LAB: Warm up: Creating passwords

(1) Prompt the user to enter two words and a number, storing each into separate variables. Then, output those three values on a single line separated by a space. (Submit for 1 point)

```
Enter favorite color:
yellow
Enter pet's name:
Daisy
Enter a number:
6
You entered: yellow Daisy 6
```

(2) Output two passwords using a combination of the user input. Format the passwords as shown below. (Submit for 2 points, so 3 points total).


```
Enter favorite color:
yellow
Enter pet's name:
Daisy
Enter a number:
6
You entered: yellow Daisy 6
```

```
First password: yellow_Daisy
Second password: 6yellow6
```

(3) Output the length of each password (the number of characters in the strings). (Submit for 2 points, so 5 points total).


```
Enter favorite color:
yellow
Enter pet's name:
Daisy
Enter a number:
6
You entered: yellow Daisy 6

First password: yellow_Daisy
Second password: 6yellow6

Number of characters in yellow_Daisy: 12
Number of characters in 6yellow6: 8
```

LAB ACTIVITY 3.16.1: LAB: Warm up: Creating passwords 0 / 5

main.py

Load default template...

```
# FIXME (1): Finish reading another word and an integer into variables.
# Output all the values on a single line
favorite_color = input('Enter favorite color:\n')


# FIXME (2): Output two password options
password1 = favorite_color
print('\nFirst password:')


# FIXME (3): Output the length of the two password options

```

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) → Output (shown be

Program output displayed here

Lab statistics and submissions Show

Solution Show

Tests Show

3.17 LAB*: Program: Painting a wall

Output each floating-point value with two digits after the decimal point, which can be achieved as follows:

```
print('{:.2f}'.format(your_value))
```

(1) Prompt the user to input a wall's height and width. Calculate and output the wall's area (integer). (Submit for 2 points).

```
Enter wall height (feet):
12
Enter wall width (feet):
15
Wall area: 180 square feet
```


(2) Extend to also calculate and output the amount of paint in gallons needed to paint the wall (floating point). Assume a gallon of paint covers 350 square feet. Store this value in a variable. Output the amount of paint needed using the %f conversion specifier. (Submit for 2 points, so 4 points total).

```
Enter wall height (feet):
12
Enter wall width (feet):
15
Wall area: 180 square feet
Paint needed: 0.51 gallons
```

(3) Extend to also calculate and output the number of 1 gallon cans needed to paint the wall. Hint: Use a math function to round up to the nearest gallon. (Submit for 2 points, so 6 points total).

```
Enter wall height (feet):
12
Enter wall width (feet):
15
Wall area: 180 square feet
Paint needed: 0.51 gallons
Cans needed: 1 can(s)
```


(4) Extend by prompting the user for a color they want to paint the walls. Calculate and output the total cost of the paint cans depending on which color is chosen. Hint: Use a dictionary to associate each paint color with its respective cost. Red paint costs $35 per gallon can, blue paint costs $25 per gallon can, and green paint costs $23 per gallon can. (Submit for 2 points, so 8 points total).

```
Enter wall height (feet):
12
Enter wall width (feet):
15
Wall area: 180 square feet
Paint needed: 0.51 gallons
Cans needed: 1 can(s)

Choose a color to paint the wall:
red
Cost of purchasing red paint: $35
```


LAB ACTIVITY 3.17.1: LAB*: Program: Painting a wall 0 / 8

main.py

Load default template...

```python
import math

# Dictionary of paint colors and cost per gallon
paint_colors = {
   'red': 35,
   'blue': 25,
   'green': 23
}

# FIXME (1): Prompt user to input wall's width
# Calculate and output wall area
wall_height = int(input('Enter wall height (feet):\n'))
print('Wall area:')

# FIXME (2): Calculate and output the amount of paint in gallons needed to paint th

# FIXME (3): Calculate and output the number of 1 gallon cans needed to paint the w

# FIXME (4): Calculate and output the total cost of paint can needed depending on c

```

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) → Output (shown be

Program output displayed here

Lab statistics and submissions **Show**

Solution **Show**

Tests **Show**

©zyBooks 03/05/20 10:22 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020