Exercise Manual for Course 458G

Python Programming Introduction

458G/MA/A.1/609/A.0

by Frank Schmidt

Technical Editor:
Alexander Lapajne

Legend for Course Icons

Standard icons are used in the hands-on exercises to illustrate various phases of each exercise.


Major step


Warning

1. ❑ **Action**


Hint


Checkpoint

Stop


Question


Congratulations


Information

Bonus


Solution/Answer

Objectives

In this exercise, you will gain experience working with Python's numeric types and arithmetic operators. To do this, you will

- **Convert string literals into numeric data types for calculations**
- **Perform integer and floating point arithmetic using variables**
- **Display formatted numeric values**

Converting a string literal into a numeric value

1. ❑ Start Eclipse if it has not already been started.

   Close any open editor panes.

   From the PyDev Package Explorer pane, open the `Ex2_1` project. From there, open the `Ex2_1.py` file for this exercise.

   The contents of the file are in the editor pane.

2. ❑ Run this program.

   You may execute the editor pane's contents by clicking the green-and-white Run button on the toolbar, or by selecting Run from the Run menu.

Hint...

Select Python Run if prompted by a pop-up dialog box.

The script's output can be viewed in the console pane in the bottom of Eclipse.

Any errors also appear in the console pane.

Look for the message `This is exercise 2.1` in the console pane to confirm the file executed.

3. ❑ For this step, you will make changes below the `Part A` comment in the source code file.

   The string assignments are provided:

   ```
   num1 = '5'
   num2 = '9'
   ```

   Convert the strings into integers and display the result of `num1 / num2`.

   Save the file and execute the program.

Hint ...

The `int()` function will convert a string to an integer. The string must contain values that can be converted to an integer.

Another hint...

`'5'` is a string representation integer and can be converted to `5`.

The strings `'5' / '9'` or `'5 / 9'` are not string representations of an integer.

The program's output appears in the console window at the bottom of the screen.

The result of the integer division `5 / 9` is `0`.

If there were errors reported in the console window, edit the source code and execute again.

You now have a working integer arithmetic calculation.

4. ❑ Convert the strings into floating point values and display the result of `num1 / num2`

Hint...

The `float()` function will convert a string to a floating point value.

Another hint...

`'5'` is a string representation of an integer and can be converted to `5.0`

The program's output appears in the console window at the bottom of the screen.

`5.0 / 9.0` is approximately `.556`.

Mixing types in arithmetic and precedence rules

Create equations to convert temperature from Celsius to Fahrenheit, and also from Fahrenheit to Celsius.

5. ❑ Make the changes below the `Part B` comment.

There are assignments to `paris_temp` and `honolulu_temp`.

The formula to convert temperature from Celsius to Fahrenheit is to multiply the Celsius temperature by the quotient of 9.0 divided by 5.0, then add 32.

The formula to convert temperature from Fahrenheit to Celsius is to subtract 32, then multiply by the quotient of 5.0 / 9.0.

Create the calculations to convert Celsius to Fahrenheit and Fahrenheit to Celsius. These calculations are to deliver floating point results.

`paris_temp` represents a Celsius value. Add statements to convert it to Fahrenheit and display the result.

`honolulu_temp` is a Fahrenheit value. Add statements to convert it to Celsius and display the result.

Hint...

Parentheses are required.

Another hint...

The subtraction must be performed first when converting Fahrenheit to Celsius.

By default, subtraction is lower precedence than division or multiplication.

`25 degrees C` is approximately `77 degrees F`.
`81 degrees F` is approximately `27.2 degrees C`.

You now have formulas to convert to either scale.

Congratulations! You have gained experience working with Python's numeric types and arithmetic operators.

If you have more time, perform additional calculations.

6. ❑ Make the following changes below the `Part C` comment.

The `price` variable is assigned.

There are three additional `discount_`*`size`* variables already assigned: `discount_small`, `discount_med`, and `discount_big`.

Hint...

`size` is used to represent a replaceable value—in this case: `small`, `med`, or `big`. The variables are named `discount_small`, `discount_med`, and `discount_big`.

Each `discount_size` variable defines a *percentage to be subtracted* from price.

Calculate and display three new `price_size` values. Each will use a different `discount_size`.

The `discount_size` is multiplied by the `price` to calculate the deduction.

Hint...

If `discount = .10`, then 10 percent is to be subtracted from `price`.

With `price = 50.00` and `discount = .10`, the adjusted `price` is `45.0`.

Another hint...

The result of 50.00 * .10 is 5. That amount would be subtracted from `price`.

Hint...

Perform these steps:

- Convert `price` to a floating point value
- Calculate the three adjusted `price_size` values after each `discount_size` has been applied
- Add `print` statements to display the floating point values after the discount has been subtracted

7. ❑ For this section, you will create your own variables, formulas, assignments, and printing. The results should be floating point values.

Here is a description of the problem to solve:

- A traveler has taken two flights
- The first flight covered 305 miles and took 62 minutes
- The second flight covered 525 miles and took 91 minutes

Add statements to perform the following calculations

- Calculate and print the speed in miles per hour for each flight
- Calculate and print the speed in kilometers per hour for each flight
- Calculate and print the average speed in miles per hour for both flights combined
- Calculate and print the average speed in kilometers per hour for both flights combined

One mile is equal to approximately 1.6 kilometers.

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.

Objectives

In this exercise, you will gain experience working with Python's string type and its operations. To do this, you will

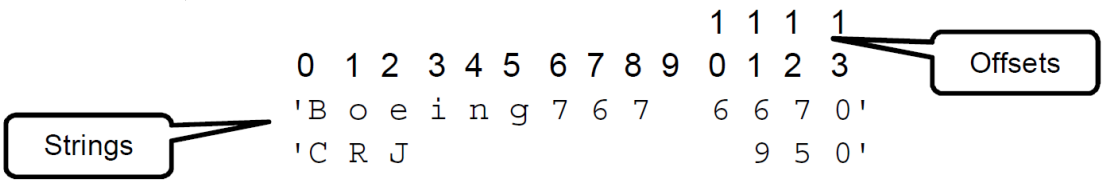
- **Use slicing techniques to extract substrings**
- **Use built-in methods to test strings**
- **Process a Comma-Separated Value (CSV) string**

Extracting string slicing and concatenation

1. ❑ Open the `Ex2_2` project. Open the `Ex2_2.py` file for this exercise.

Beneath the `Part A` comment are two `planeN` variables that have been assigned fixed-length strings.

The first value is the plane type. The second is its flight range in miles. Their offsets into the strings are described in the following diagram.



The notation `N` is used to describe a number. The variables are `plane1` and `plane2`.

2. ❑ Display the plane type and flight range for each `plane`*`N`* string.

Hint...

Use string slicing to extract type and range for each `plane`*`N`*.

Another hint...

An unbounded slice terminates at the end of the string.

3. ❑ Continue working below the `Part A` comment. Display the concatenation of the two plane types and the sum of the two plane ranges.

Hint...

The same operator is used in both statements.

Another hint...

Strings must be converted to a numeric type for addition.

Creating CSV strings

4. ❑ Continue working below the `Part A` comment.

CSV stands for comma-separated value. It is a common import/export format for database tables or spreadsheets.

Create and display a comma-separated string of both planes' types and ranges. Use the `format()` function to create the CSV strings.

Hint...

The string to be formatted may contain normal text as well as format specifications.

Another hint...

Include the `,` between the {*spec*} within the string.

Your syntax may resemble:

```
print '{0} , {1} , {2} , {3}'.format( ...
```

The spaces arouned each comma are optional.

Using string methods

5. ❑ Add the new statements below the `Part B` comment. There are two `plane`*N* variables that have been assigned variable-length, comma-delimited strings.

   Use the `find()` method to discover the offset of the `','` within each string.

   Use string slicing, and the offset value discovered above, to display the type and range for each `plane`*N* assigned.

   The first field is the plane type; the second is its range in miles.

Hint...

Use a string method to locate the offset of the comma within the string.

Solution...

```
plane1.find(',')
```

Testing and branching using `if`

6. ❑ Add statements below the `Part C` comment and use the variables created below `Part B`.

   Test each plane's type and display a message if the type is completely uppercase.

Hint...

The string method `isupper()` will be used.

Congratulations! You have gained experience working with Python's string type, string operations, and conditional tests.

If you have more time: Adding more conditionals

7. ❑ Use more `if` statements to:
   - Test whether a plane type ends with a digit
   - Determine which plane has the greater range

Hint...

The string method `isdigit()` will be used on a slice of the last character of the string.

The range should be converted into a numeric type before the comparison.

Using quotation marks

8. ❑ For this step, use the two print statements below the `Part D` comment. You will have to add the strings to produce the desired output.

All `print` output should contain the quotation marks as described.

9. ❑ Use the first `print` statement to display the following:

```
Python is Guido's invention
```

Hint...

You will need some type of quotation mark to print the assigned single quote textually.

10. ❑ Use the second `print` statement to display the following:

```
They say, "Python is Guido's invention."
```

11. ❑ Continue adding the new statements at the end of the file. There are five variable assignments, `airportN`, of CSV strings. The first field is the airport code; the second is the city name.

    Create and display a single, new CSV string of all airport codes. Each airport code should be in double quotation marks.

    The output should look similar to:
    `"HNL","LHR","ARN","HKG","GCM"`

Hint...

Use slicing to identify each airport code from its CSV string. The airport code preceeds the offset of the `","` comma.

Use the `format()` function to merge the airport codes into the new CSV string.

12. ❑ Create and display a single, new CSV string of all city names. Each city name should be in double quotation marks.

    Using a loop would help with repetitive steps. Loops are discussed in Chapter 3.

13. ❑ Display the result of the `split()` method applied to `airport1`.

    `split()` returns a list. List processing is described in Chapter 3.

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.

Objectives

In this exercise, you will gain experience applying loops and built-in methods to manage collection types.

- **Use slicing techniques to unravel a sequence**
- **Use built-in methods to manage lists and tuples**

Slicing a list

1. ❑ Open the `Ex3_1` project and the `Ex3_1.py` file for this exercise.

   Make the first set of changes below the `Part A` comment. The variable `codelist` has been assigned a list of three-letter airport codes.

   Print the first two codes and the last two codes.

Hint...

An unbounded slice would be helpful.

The slice of `[-1:]` references from the final index of the list.

Another hint...

A slice of `[-2:]` references the final two indices.

2. ❑ The variable `flightlist` has been assigned a list containing the details of an airline flight.

   `[0]` is the `departcity`
   `[1]` is the `arrivecity`
   `[2]` is the `departtime`
   `[3]` is the `departday`
   `[4]` is the `arrivetime`
   `[5]` is the `arriveday`
   `[6]` is the `cost`
   `[7]` is the `code`

   There are comments within the file describing the mapping. These should be uncommented and used to assign the 8 values.

   Print the `departcity`, `arrivecity`, `departday`, and `arriveday`.

3. ❑ Use the sequence unpacking to assign the list contents to separate variables. Display the `departcity`, `arrivecity`, `departday`, and `arriveday`.

Hint...

Sequence unpacking will require eight variable names.

Applying list methods

4. ❑ Reverse the contens of `codelist`, then display `codelist`.

5. ❑ Sort `codelist` in ascending order, then display `codelist`.

A list is mutable; it can be changed in place.

Hint...

The `sort()` and `reverse()` functions return `None`. The original list is sorted.

Testing shared references

6. ❑ Add the following assignment:

```
aptlist = codelist
```

Execute the `pop()` function on `aptlist`, then display both `aptlist` and `codelist`.

Hint...

The syntax is *`list_name`*`.pop()`.

The returned value can be ignored.

The last element referenced by both names is gone.

7. ❑ Add an `if` test using the `is` operator to test for a shared reference. Display some messages to indicate whether a shared reference exists or not.

Assignment creates a shared reference.

Congratulations! You have managed lists and slices of lists.

If you have more time, explore more list handling.

8. ❑ The `list()` function can duplicate a list.

Assign a copy of `codelist` to `aptlist`.

Hint...

A new list is returned by the `list()` function.

9. ❑ Execute the `pop()` function on `aptlist`, then display both `aptlist` and `codelist`.

Only the object referenced by `aptlist` was affected.

10. ❑ Another way to copy a list is to assign a slice of the entire list.

Using slicing, assign a copy of `codelist` to `aptlist`.

11. ❑ Test if the two lists have identical contents and display messages to indicate whether there was equality in contents of the lists or not.

12. ❑ Test if the two lists reference the same objects and display messages to indicate whether there was a shared reference, or not.

Hint...

The == operator is used to test equality.

The `is` keyword is used to test a shared reference.

The lists have equal values but are not shared references.

13. ❑ Concatenate `codelist` to itself and assign the result to `catlist`.

Repeat `codelist` twice and assign the result to `repeatlist`.

Hint...

The list concatenation and repetition operators are the same for strings or lists.

Another hint...

Use + to concatenate lists. The result is a new list.

Use * to repeat lists. The result is a new list.

14. ❑ Test if the two new lists have identical contents and display messages to indicate whether there was equality in contents of the lists or not.

These new lists have equal values but are not shared references.

Additional list modification methods

15. ❑ Extend `catlist` by placing `'ABC'` before the first element of the list.

16. ❑ Extend `repeatlist` by placing `'XYZ'` after the last element of the list.

Hint...

The `insert()` and `append()` functions may be used.

17. ❑ Test whether `catlist` and `repeatlist` are the same length. Display a message to indicate whether they are the same length or not.

18. ❑ Test whether `catlist` and `repeatlist` are equal in value. Display a message to indicate whether they are equal or not.

19. ❑ Display both `catlist` and `repeatlist`.

The two lists are the same length.

20. ❑ Remove the *first* element from `catlist`.

21. ❑ Remove the *last* element from `repeatlist`.

22. ❑ Verify that the two lists are now equal in value.

Hint...

The `pop()` function can remove list elements.

Comparing lists and tuples

23. ❑ Convert `repeatlist` to a tuple.

24. ❑ Test whether the length of `repeatlist` remains equal with the length of `catlist`.

25. ❑ Test whether the value of `repeatlist` remains equal with the value of `catlist`.

The length is the same; the value is not the same.

26. ❑ Attempt some of the previous list methods to `repeatlist`. Try the `append()`, `sort()`, or `pop()` methods.

These will fail. Tuples are an immutable type.

Additional list handling methods

27. ❑ Sort `repeatlist`, then display the new values as a tuple.

Hint...

Convert the tuple to a list for modifications. Convert the list back into a tuple after modifications.

Another hint...

Use the `list()` and `tuple()` functions.

Which methods performed in the bonus part of this exercise would not work with a tuple?

_____

_____

Solution...

`pop()`, `insert()`, `append()`, `sort()`, `reverse()`, `remove()`

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.

Objectives

In this exercise, you will gain experience applying loops and built-in methods to manage collection types.

- **Access the keys and values of a dictionary**
- **Perform membership testing**
- **Loop through the contents of a collection**

Looping through a dictionary

1. ❑ Below the `Part A` comment, the variable `city_code_dict` has been assigned.

   The dictionary keys are the three-letter airport codes. The dictionary values are the city names.

   Use the `keys()` method and a loop to display dictionary keys.

2. ❑ Use the `values()` method and a loop to display the dictionary values.

Hint...

Both `keys()` and `values()` return lists.

Another hint...

A `for` loop will iterate through a sequence like a list.

3. ❑ Create a third loop that iterates through the dictionary keys *without* using a method. Display each key and its associated value.

Hint...

Use the dictionary name without a method for access to the keys.

Use the dictionary keys to access the associated values.

Another hint...

A sample code layout may contain:

```
for key in dictionary:
  print key, dictionary[key]
```

Membership testing using loops and `if`

4. ❑ Below the `Part B` comment, the variable `codelist` has been assigned a list of airport codes.

   Use a `for` loop, `if` test, and `in` keyword to determine which values in `codelist` are keys in `city_code_dict`.

   Create a list of the values that are keys and another list of the values that are not keys.

5. ❑ Display both lists.

Hint...

A sample coding layout may contain:

```
for value in list:
  if value in dictionary:
```

`['HNL', 'ITO', 'LHR', 'GCM']` is the list of keys.

`['LGA', 'MSY']` are not keys.

Membership testing using list comprehensions

6. ❑ Use list comprehensions to:
   - Display a list of the values from `codelist` that are keys in `city_code_dict`
   - Display a list of the values from `codelist` that are not keys in `city_code_dict`

Hint...

Two list comprehensions are required.

Another hint...

Use one list comprehension to determine which values are keys in the dictionary.

Use a second list comprehension to determine which values are not keys in the dictionary.

`['HNL', 'ITO', 'LHR', 'GCM']` is the list of keys.

`['LGA', 'MSY']` are not keys.

Membership testing using the set approach

You can compare the contents of two collections to find the common members and differing members without loops or conditionals by using set operations.

7. ❑ Determine which values from `codelist` are keys in `city_code_dict` by using set operations:
   - Display a list of the values that are keys
   - Display a list of the values that are not keys

`['HNL', 'ITO', 'LHR', 'GCM']` is the list of keys.

`['LGA', 'MSY']` are not keys.

Hint...

The `set()` function returns a set from the sequence.

The intersection operator `&` will deliver a set of the common members. The difference operator `-` will deliver the differing members.

Congratulations! You have used loops, sets, list comprehensions, and membership testing to compare collections.

If you have more time, perform additional testing with more complex collections.

More membership testing using loops and `if`

8. ❑ Below the `Part B` comment, the variable `flightlist` is assigned.

The `[0]` and `[1]` elements of `flightlist` are the airport codes for the departure airport and the arrival airport. These values will be compared to the keys of `city_code_dict`.

Determine if *both* elements of `flightlist` are also keys in `city_code_dict`.

Display a message indicating whether both codes are keys or not.

Hint...

A compound conditional will be required.

Below the `Part C` comment, a variable `flightdict` has been assigned.

The dictionary key is the flight number. The value is a list. Each list describes the flight details. The list contents correspond to the same flight information used in Exercise 3.1.

The list's mapping is:

- *`[0]` is the `departcity`*
- *`[1]` is the `arrivecity`*
- *`[2]` is the `departtime`*
- *`[3]` is the `departday`*
- *`[4]` is the `arrivetime`*
- *`[5]` is the `arriveday`*
- *`[6]` is the `cost`*
- *`[7]` is the `code`*

9. ❑ Create and display a list of round-trip flights and a list of overnight flights.

   Print these two lists.

Hint...

A round-trip flight has the same value for `departcity` and `arrivecity`.

An overnight flight has different values for `departday` and `arriveday`.

The list of flight numbers for the round-trip flights is `[132, 390, 1572]`.

The list of flight numbers for the overnight flights is `[276, 498, 444]`.

10. ❑ Solve the same problem by using list comprehensions. Display the lists of overnight flights and round-trip flights.

The list of flight numbers for the round-trip flights is `[132, 390, 1572]`.

The list of flight numbers for the overnight flights is `[276, 498, 444]`.

11. ❑ Display the flight numbers and flight information from `flightdict` sorted by flight number.

`102` is the lowest flight number; `1572` is the highest.

Hint...

The `keys()` method returns a list.

12. ❑ Below the `Part D` comments, there are five variable assignments, `airportN`, of CSV strings. The first field is the airport code; the second is the city name.

Create and display a single, new CSV string of all airport codes. Each airport code should be in double quotation marks.

Create and display a single, new CSV string of all city names. Each city name should be in double quotation marks.

The output should look similar to:

```
"HNL","LHR","ARN"...
"Honolulu","London/Heathrow","Stockholm/Arlanda"...
```

Hint...

Use slicing to extract each airport code from its CSV string.

The `format()` function may be used to add the double quotes within the strings.

The `join()` function can construct a delimited string from a list.

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.

Objectives

In this exercise, you will gain experience creating and calling a function, passing arguments, and capturing the function's returned value.

- **Create a function using the `def` statement**
- **Call a function passing in an argument list**
- **Return results from functions**

Creating a function

1. ❑ Open the `Ex4_1.py` project. Open the `Ex4_1.py` file for this exercise.

   There are two variables assigned near the top of the file, `city_code_dict` and `flightdict`, for use in this exercise.

   `city_code_dict` is a dictionary of airport information. The key is the airport code and the value is the city.

   `flightdict` is a dictionary of flight information. The key is the flight number and the value is a list of flight information. The list contents correspond to the same flight information used in previous exercises:

   The list's mapping is:
   `[0]` is the `departcity`
   `[1]` is the `arrivecity`
   `[2]` is the `departtime`
   `[3]` is the `departday`
   `[4]` is the `arrivetime`
   `[5]` is the `arriveday`
   `[6]` is the `cost`
   `[7]` is the `code`

2. ❑ Create a function named `list_all_cities()` that displays the three-letter airport code and the corresponding airport name for all of the entries of the global `city_code_dict` dictionary.

   This function will accept no parameters and return no value.

Hint...

The `def` statement is required.

The function body must be indented.

Functions must be defined above their calls within the same file. The functions will encapsulate the same type of coding created in Exercise 3.1.

3. ❑ Add function definitions below the `Part A` comment.

4. ❑ Add function calls below the `Part B` comment.

Hint...

A dictionary method can `return` the keys or values of the dictionary as desired.

This function will use the global `city_code_dict` dictionary.

The function is now complete.

Calling a function

5. ❑ Below the `Part B` comment, add the statement to execute the function.

Hint...

Remember to use `( )` on the function call.

You have written and called a function.

Passing arguments to a function by position

6. ❑ Create a function named `flights_per_city()` that displays flight information for flights that fly *from* a particular city.

   The function will receive one argument, a three-letter airport code. It will use the global variable `flightdict`.

Hint...

The `departcity` is the first element of each list within the `flight_dict` dictionary values.

A parameter should be specified within the function's `def` statement.

Within the function, a loop is required to access each element of `flightdict`. A test is required to compare the parameter with the proper list element.

7. ❑ Display the flight number and all of the flight details if the parameter matches a flight's `departcity`.

The function is now complete.

8. ❑ Below the `Part B` comment there are three assignments to the variable `searchcity`, each assigning a different airport code.

   Add the calls to `flights_per_city()` three times, each time with a different airport code.

Hint...

For the first call, the argument is `HNL`.
For the second call, the argument is `CUR`.
For the third call, the argument is `ITO`.

Congratulations! You have created and called functions.

If you have more time, `return` a value from a function.

9. ❑ Create a new function, `flights_per_cities()`, that will search flights that fly *from* a particular airport and *to* a particular airport.

   This new function will have two positional parameters. A three-letter airport code for the *from* airport is the first. A three-letter airport code for the *to* airport is the second.

   The global variable `flightdict` will be used again.

10. ❑ Return a list of all flight numbers for flights with a `departcity` and `arrivecity` that match the parameters.

Hint...

Each dictionary value is a list. The `departcity` airport is element `[0]` of the list and the `arrivecity` is element `[1]`.

Another hint...

A list comprehension will be helpful.

11. ❑ Add the `return` statement to the end of the function. It should return the completed list of flight numbers.

This function is now complete.

12. ❑ Two variables have been assigned.

```
departcity =  'NRT'
arrivecity = 'ITO'
```

Use these as arguments to `flights_per_cities()`, then display the returned list.

Flight number 498 travels between these two cities.

Using keyword parameters

13. ❑ Add the following two assignments:

```
departcity = 'HKG'
arrivecity = 'HNL'
```

14. ❑ Examine the `def` statement of `flights_per_cities()` and note the parameter names.

15. ❑ Add a new call to `flight_per_cities()` passing the new variables as keyword arguments. Display the returned list.

Hint...

Use assignments to the parameters' names as specified in function header:

```
def flights_per_cities( param1, param2)
```

Flight number 375 travels between these two cities.

16. ❑ Create a new function, `discount()`, to calculate and return the `price` of a flight after a `discount` has been applied.

    A `discount` is a percentage of the price to be subracted. If `price` is `10` and discount is `0.2`, the new price is `8.0`.

    Use the following pairs as the arguments:

```
price = 100    disc = 0.05
price = 299    disc = 0.15
price = 399.95 disc = 0.10
```

    Display the `price` before and after the discount is applied.

    Put the call to `discount()` within the `print` statement.

Hint...

The function body will contain only the calculation.

It could be:

```
return price - ( price * disc )
```

Function calling a function

17. ❑ Extend the previous solution step by creating a new function, `discount_printer()`. Call the new function with the two lists described below as arguments:

```
pricelist = [100, 299, 399.95]
disclist = [0.05, 0.15, 0.10]
```

These two lists are assigned in a particular order. Offset `[0]` of `pricelist` corresponds with `[0]` of `disclist`.

The price of `100` receives a discount of `0.05`. The price of `299` receives a discount of `0.15`, etc.

From within `discount_printer()`, call `discount()`, passing the `pricelist` and `disclist` pairs as arguments.

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.

Objectives

In this exercise, you will gain experience with two special type functions: single-statement `lambda` functions and a generator function that behaves as an iterator.

- **Create a `lambda` function**
- **Create a generator function**

Creating a `lambda` function with one argument

1. ❑ Open the `Ex4_2` project and `Ex4_2.py` file for this exercise.

   Add statements for this step below the `Part A` comment to create a dictionary named `temp_converter`.

   - Assign the string `'ctof'` as the first dictionary key. The corresponding dictionary value is a `lambda` function to convert a Celsius temperature to the Fahrenheit equivalent.
   - Assign the string `'ftoc'` as the next dictionary key. The corresponding dictionary value is a `lambda` function to convert a Fahrenheit temperature to the Celsius equivalent.

Hint...

The dictionary will be assigned with syntax similar to:

```
temp_converter = {'ctof': lambda .... ,
                  'ftoc': lambda ... }
```

The formula to convert C to F is:
`F = C * 9.0 / 5.0 + 32`

The formula to convert F to C is:
`C = (F - 32) * 5.0 / 9.0`

Hint...

Identifiers immediately after the keyword `lambda` are the function's parameters.

The result of any calculation within the function is returned.

Another hint...

A sample `lambda` function to calculate the area of a circle may contain:

```
lambda radius:  3.14 * radius ** 2
```

2. ❑ The assignments to `paris_temp` and `honolulu_temp` are provided for testing.

   `paris_temp` represents a Celsius temperature. Use the `lambda` functions within `temp_converter` and display this value in both scales.

3. ❑ `honolulu_temp` represents a Fahrenheit temperature. Use the `lambda` functions within `temp_converter` and display this value in both scales.

`25 degrees C` is approximately `77 degrees F`.
`81 degrees F` is approximately `27.2 degrees C`.

You now have formulas to convert to either scale.

Generator functions

4. ❑ Below the `Part B` comment, there is a function named `nextid()`. The function will be used to create a string to identify an airline reservation.

   Review the coding provided in the function and ask your instructor for help if needed.

Hint...

The function:

- Accepts a single parameter named `start`
- Assigns a 24-character string to `resletters` and `0` to `residex`
- Displays `start` after converting to a string, concatenated with a single element from `resletters`
- Increments `start`
- Tests `residex` and either increments it or resets it to `0`

Another hint...

The function currently `returns` no value and does not maintain its state between executions.

5. ❑ Convert the `nextid()` into a generator function that will deliver a new string with each call.

Two changes are required:

1. Add a `while True:` header statement after the third line of the function
   - Add it after the `residex = 0` line
   - Indent the statements that follow so they are part of this `while` loop
2. Replace the `print` statement with a `yield` statement
   - `yield` makes this function a generator
   - The same value that was `printed` in now `yielded`

Hint...

The partial coding may include:

```
def nextid(start):
  resletters = 'ABCDEFGHJKLMNPQRSTUVWXYZ'
  residex = 0
  while True:
    yield str(start) + resletters[residex]
    ...
```


6. ❑ Assign a reference to the new generator function. Replace the current function call, `nextid(99)`, with `reservation = nextid(99)`.

`reservation` is now an iterable object.

7. ❑ Create a `for` loop to call the new function 30 times.

8. ❑ Within the `for` loop, `print` each `yielded` value.

Hint...

`range(30)` will deliver a sequence of integers from 0 through 29.

`next(reservation)` will deliver each `yielded` value.

The values yielded are `99A`, `100B`, `101C`, `102D`, . . . `123A`, `124B`, `125C`, `126D`, `127E`, `128F`.

Congratulations! You have called and used `lambda` functions and generator functions.

If you have more time, extend `temp_converter` to contain additional `lambda` functions.

9. ❑ Add two new elements to `temp_converter` to convert from Celsius to Kelvin and from Kelvin to Celsius.
   - To convert Celsius to Kelvin, use the formula `K = C + 273.15`
   - To convert Kelvin to Celsius, use the formula `C = K - 273.15`

10. ❑ Add a new assignment, `moon_temp = 36`.

The assignment to `moon_temp` represents a Kelvin value.

11. ❑ Use the modified dictionary to display:
    - `paris_temp` in Celsius, Fahrenheit, and Kelvin
    - `honolulu_temp` in Celsius, Fahrenheit, and Kelvin
    - `moon_temp` in Celsius, Fahrenheit, and Kelvin

Hint...

There is no need to convert Kelvin directly to Fahrenheit. Simply convert Kelvin to Celsius, then Celsius to Fahrenheit using the existing functions.

There is no need to convert Fahrenheit directly to Kelvin. Simply convert Fahrenheit to Celsius, then Celsius to Kelvin using the existing functions.

Another hint...

A value returned from one function can be used as the argument. For example, to use a value returned from *`fun1()`* as an argument to *`fun2()`*, use:

```
fun2(fun1(args))
```

25 degrees C is approximately 77 degrees F and approximately 298 degrees K.

81 degrees F is approximately 27.2 degrees C and approximately 300 degrees K.

36 degrees K is approximately -237 degrees C and approximately -395 degrees F.

12. ❑ Below the `Part C` comment, a small dictionary named `city_fees_dict` is assigned. The dictionary key is `'HNL'` and the value is a `lambda` function.

    There are also two assignments to `price` and `tax`.

    To test this dictionary, construct a small `for` loop to:
    - Access each key within `city_fees_dict`
    - Call its `lambda` function
    - Display the returned value

    For `'HNL'`, the fee returned is `20.0`.

13. ❑ Extend the `city_fees_dict` dictionary by adding the following key–value pairs:

Key	Calculation
`'ITO'`	Same as key `'HNL'`
`'LHR'`	`price * tax + 100`
`'ARN'`	Same as key `'LHR'`
`'HKG'`	`price * tax + 150`
`'CDG'`	Same as key `'LHR'`

14. ❑ Test the dictionary to verify each calculation works.

    A dictionary of `lambda` functions has been used.

A list of `lambda` functions

Beneath the `Part D` comment, two lists are assigned.
- *`shapelist` is assigned a series of strings that are names of shapes*
- *`arealist` is assigned a series of `lambda` functions that calculate the area of various shapes*

For each shape name in `shapelist`, its corresponding area calculation is in `arealist`.

Hint...

For example, the string `'circle'` and the formula to calculate the area of a circle are both at offset `[3]` within their respective lists.

15. ❑ Add a loop to display each shape name and the result of the calculation for that shape.

Hint...

A `for` loop and `range()` function may be helpful.

The `lambda` functions are called without arguments.

Another hint...

The `range()` function is needed for the offset values.

The area of the `'square'` is `9.0`.

The area of the `'rectangle'` is `13.5`.

The area of the `'triangle'` is `6.75`.

The area of the `'circle'` is `19.625`.

Hint...

Each element of the list is a string.

16. ❑ Combine the two lists into a dictionary. The shape name is the key and the `lambda` function is the value.

Display the keys and associated calculated areas.

Hint...

The `zip()` function combines two sequences into a list of tuples.

The `dict()` function returns a dictionary from a sequence of key–value pairs.

The `lambda` function testing is complete.

Another generator function

17. ❑ The Fibonacci sequence is a series of numbers in which each value is the sum of the previous two values from the sequence. For example:

    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

    Create a generator function that will `yield` the sum of the previous two values. The parameters will provide the two values to start the sequence.

    For example, if the parameters were `(3, 6)`:

    3, 6, 9, 15, 24, 39, 63, 102, 165, ...

Hint...

The function will accept two parameters and `yield` a single value.

Another hint...

The previous two numbers must be kept between each call.

18. ❑ Create an iterable object using `(3, 6)` as the argument list. Call the generator function 10 times.

The values are: 3, 6, 9, 15, 24, 39, 63, 102, 165, 267, 432, and 699.

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.

Objectives

In this exercise, you will gain experience creating new classes that define data used for an airline. The new classes will contain constructor methods.

- **Create classes**
- **Add a constructor method**

Creating classes and using `__init__()` functions.

1. ❑ Open the `Ex5_1` project and the `Ex5_1.py` file for this exercise. Make the following changes below the `Part A` comment.
   - Create a new base class named `Trip` with these attributes:
     - `departcity`
     - `arrivecity`
     - `departtime`
     - `departday`
     - `arrivetime`
     - `arriveday`
   - Assign `None` as a default for all attributes in the constructor's parameter list.

Hint...

Remember to use `self`.

A general layout of the code might be:

```
class Trip:
  def __init__(self, departcity=None, arrivecity=None, …):
    self.departcity = departcity
    self.arrivecity = arrivecity
    …
```

The first class has been created.

2. ❑ Below the `Part A` comment, there are some assignments to be used to initialize a `Trip` instance.

   Use the variables `depcity`, `arrcity`, `deptime`, `depday`, `arrtime`, and `arrday` as the argument list when creating the instance.

3. ❑ Create a new `Trip` instance and display the attributes of the instance.

Hint...

The variable names are very similar to the parameter names expected in `__init__()`.

Use a keyword style argument passing when constructing the instance.

Another hint...

The coding may contain some of the following:

```
mytrip = Trip(departcity=depcity,
arrivecity=arrcity, departtime=deptime, ...
```

An instance of class `Trip` has now been created.

4. ❑ Below the `Part B` comment, two list variables are assigned: `hawaiilist` and `cariblist`. These will become class variables.

   Using the editor, move these assignments and add indentation so that they become class variables within the `Trip` class.


Hint...

Move the list assignment below the body of the constructor method within the `Trip` class definition.

Indent to keep these assignments within the class, but *not* within the constructor method.


`hawaiilist` and `cariblist` are class variables.


Warning! These two lists are needed for class methods created later. Be sure that they are within the `Trip` class definition.

5. ❑ Outside of the `class` statement, display `cariblist` and `hawaiilist`.


Hint...

Use the class name to access these class variables.


Solution...

```
Trip.hawaiilist
Trip.cariblist
```


Adding a method

6. ❑ For this step, continue to add new statements within the `Trip` class definition.

Add a new method within the `Trip` class named `is_round_trip()`. This method tests for a round-trip.


If the `departcity` and the `arrivecity` are equivalent, the trip is a round-trip.

This method will return a Boolean indicating whether a trip is a round-trip or not.

Hint...

Compare `self.departcity` to `self.arrivecity`.

7. ❑ Using the instance created in Step 3 above, determine and display whether that was a round-trip.

The trip from `'CUR'` to `'HNL'` was not a round-trip.

Congratulations! You have created and tested a class that defines data used for an airline.

The new class contains an `__init__()` constructor method, some class variable, and an additional method.

If you have more time, add methods.

Warning! Be sure that your program works up to this point. The following steps will continue building on this work.

Hint...

If you need help, open the `Ex5_1_EndPoint.py` solution file for a working `class Trip` statement and instance creation.

8. ❑ For this step, add the following methods within the `Trip` class definition:
   - `is_over_night()`: will return `True` if the `departday` is not equal to the `arriveday`
   - `is_hawaiian()`: will return `True` if the `arrivecity` is contained within `Trip.hawaiilist`
   - `is_caribbean()`: will return `True` if the `arrivecity` is contained within `Trip.cariblist`
   - `is_interisland()`: will return `True` if both the `arrivecity` and `departcity` are contained within `Trip.hawaiilist`

The class now has four additional methods.

9. ❑ Below the `Part B` comment are some comments that contain assignments to several `trip`*N* objects.

   Use the assignments to construct the four `Trip` instances. Test the data by displaying the four `trip`*N*`.departcity` values.

Hint...

> `trip`*N* refers to a numbered identifier: `trip1`, `trip2`, `trip3`, and `trip4`.

10. ❑ Run the program to verify that there are no errors.

11. ❑ Further below the `Part B` comment, a list and function are provided in the comments to use for testing:

```
#triplist = [trip1, trip2, trip3, trip4]
#def print_trip(t):
#    print ...
```

   Uncomment this coding to use the list and function to display all attributes of the `trip`*N* objects.

You now have a list and a function to aid testing of the `tripN` objects.

12. ❑ Add a loop to display and test each `Trip` instance within `triplist`. Within the loop:
    - Call `print_trip()` to display the attributes
    - Call `is_round_trip()` and display a message if it was a round-trip
    - Call `is_caribbean()` and display a message if it was a Caribbean trip
    - Call `is_hawaiian()` and display a message if it was a Hawaiian trip
    - Call `is_over_night()` and display a message if it was an overnight trip
    - Call `is_interisland()` and display a message if it was an interisland trip

Hint...

Some of the coding may contain:

```
for t in triplist:
    print_trip(t)
    if t.is_round_trip:
        print 'is RoundTrip'
    ...
```

`trip1` is Hawaiian and interisland.

`trip2` is Hawaiian.

`trip3` is Caribbean.

`trip4` is overnight.

Additional classes

13. ❑ Add statements below the `Part A` comment to create two additional classes with constructor methods.
    - The `Aircraft` class has two attributes: `code` and `name`
    - The `Airport` class has two attributes: `citycode` and `city`

14. ❑ Assign a default value `None` to all parameters in the constructor's `def` statement.

Hint...

Classes are usually created at the top of the file.

The two new classes are completed.

15. ❑ Below the `Part C` comment are some additional comments that describe some sample data for `Aircraft` and `Airport` objects.

    Test the two new classes by creating instances and displaying attributes.

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.

Objectives

In this exercise, you will gain more experience using classes by creating a subclass that inherits from its base class. The new subclass will contain additional attributes and methods.

- **Create subclasses**
- **Extend subclasses**
- **Verify inheritance**

Creating subclasses

The exercise will build on your work from Exercise 5.1. Some of the previous coding has been provided.

Warning! A working `Trip` class with its five methods is required for this exercise.

1. ❑ Open the `Ex5_2` project and the `Ex5_2.py` file.

The `Trip` class is provided.

2. ❑ Examine the `Trip` class definition. Review the:
   - Class definition
   - `__init__()` constructor
   - Class attributes: `hawaiilist` and `cariblist`
   - Five methods: `is_round_trip()`, `is_caribbean()`, `is_hawaiian()`, `is_overnight()`, `is_interisland()`.

   Ask your instructor for an explanation of any coding you do not understand.

3. ❑ Add statements below the `Part B` comment.

   Create a subclass of `Trip` named `Flight`. The new class contains has three additional attributes: `flightnum`, `cost`, and `code`.

   The new `Flight` class will also have its own `__init__()` constructor with keyword parameters.

   Provide these defaults for the keyword parameters:
   - `flightnum = -1`
   - `cost = 0.0`
   - `code = 0`
   - For all other parameters, assign `None`

   Use the `super()` function within `Flight`'s constructor to initialized the `Trip` class attributes.

   
   The `code` attribute defines an `Aircraft` code, the type of plane used for this flight.


Hint...

The `Flight.__init__()` method parameter list contains `Flight` and `Trip` attributes.


Another hint...

Within `Flight.__init__()` use `super(Flight, self).__init__( ...`

4. ❑ Add an additional method called `discount()` within the `Flight` class. It calculates a discount and changes the instance `cost` attribute. The discount for a particular flight is based on its `departcity` and `arrivecity` values.

   The inherited `Trip` methods `is_interisland()`, `is_hawaiian()`, and `is_caribbean()` will be called from `discount()` to determine if a particular flight qualifies for a discount.

The discount reductions are:

- *If `is_interisland()` returns `True`, reduce `cost` by 5 percent*
- *If `is_hawaiian()` returns `True`, reduce `cost` by 10 percent*
- *If `is_caribbean()` returns `True`, reduce `cost` by 15 percent*

It is possible that a single flight may pass both the `is_interisland()` and `is_hawaiian()` tests. If so, only the smaller discount is applied.

The new subclass has been created.

Creating an instance and inheriting attributes

5. ❑ Below the `Part C` comment, the test data has been provided within comments.

   Uncomment the six assignments to the `flight`*`N`* object.

6. ❑ Perform the following steps for each `flight`*`N`* object:
   - Display the `flightnum` and original `cost`
   - Call the `discount()` method
   - Display the `flightnum` and new `cost`

Flights 204, 336, 660, and 681 were discounted.
Flights 102 and 753 were not discounted.

Adding another subclass

7. ❑ Below the `Part B` comment, add statements to create a new subclass of `Trip` named `Cruise`.

   It has some additional attributes: `cruisenum`, `cost`, and `code`.

   Provide these defaults for the constructor's keyword parameters:
   - `cruisenum = -1`
   - `cost = 0.0`
   - For all other parameters, assign `None`

   Use the `super()` function within `Cruise`'s constructor to initialized the `Trip` class attributes.

The `code` attribute references either `'I'` or `'O'` to indicate "Inside" or "Outside" cabin.

8. ❑ The `Cruise` class has its own constructor and `discount()` method to reduce the `cost` attribute.

   Call
   the `Trip` method's `is_interisland()` and `is_hawaiian()` from `discount()`.
   Deduct the following discount percentages:
   - If `is_interisland()` returns `True`, reduce `cost` by 10 percent
   - If `is_hawaiian()` returns `True`, reduce `cost` by 20 percent

Hint...

The `Cruise __init__()` function should call
the `Trip.__init__()` constructor by using `super()` as was done in the `Flight` class. For example:

```
super(Cruise, self).__init__( ...
```

The new class has been added.

9. ❑ Below the `Part D` comment, a series of `Cruise` objects have been assigned to `cruiselist`.

   Uncomment this block of statements to create the collection five new `Cruise` instances.

10. ❑ Perform the following steps for each `Cruise` object:
    - Display the `cruisenum` and original `cost`
    - Call the `discount()` method
    - Display the `cruisenum` and new `cost`

All five `Cruise` objects were discounted.

Congratulations! You have added subclasses that contain additional attributes and methods. You have created multiple objects from these new subclasses and applied their methods.

If you have more time: Adding more methods

11. ❑ Within the `Trip` class, add a method named `print_arrival_warning()` that will `print` the string `'Arrive 1.5 hours early for a trip'`, followed by the `arrivecity` attribute.

12. ❑ Within the `Cruise` class, add a method named `print_arrival_warning()` that will `print` the string `'Arrive 3 hours early for a cruise'`, followed by the `arrivecity` attribute.

The `Flight` objects will have access to the method within `Trip` due to inheritance.

13. ❑ Loop through a list of all `Flight` and `Cruise` objects. Execute each object's `print_arrival_warning()` method.

Adding a new class

Hint...

Be careful to *not* assign a list within a list with syntax similar to:

```
list1 = [1, 2, 3]
list2 = ['a', 'b']
biglist = [list1, list2]  # list of lists
```

Another hint...

Use concatenation:

```
list1 = [1, 2, 3]
list2 = ['a', 'b']
biglist = list1 + list2
```


14. ❑ The function `nextid()` was written for Exercise 4.2. It is a generator function that delivers successive values.

    Copy your own from the `Ex4-2.py` file. A working copy can also be found in the `Ex4_2_EndPoint.py` file.

    Using the editor, copy the `nextid()` function into the current file near the `Part D` comment. Copy its initial call, `reservation = nextid(99)`, as well.

    The `nextid()` function is not part of any class. It is a standalone, or global, function.

Hint...

Your file should contain:

```
def nextid(start):
    resletters = ...
...

reservation = nextid(99)
# Part D
```

Another hint...

`reservation` is the iterable object.

The function has been added.

15. ❑ Create a new class named `Reservation` below the existing `Trip`, `Flight`, and `Cruise` classes.

`Reservation` contains three attributes: `name`, `reservationid`, and `flightref`.

- The `name` attribute is a string of the passenger's name
- The `reservationid` attribute is string that is a unique identifier
- The `flightref` attribute is a reference to a `Flight` object; for example, `flight1`

Include a test in `Reservation`'s constructor method:

- If the `flightref` parameter is `None`, assign `self.flightref` from `nextid()`
- Else assign `self.flightref` from its parameter

Hint...

A partial layout of the coding might contain:

```
class Reservation( ...):
    def __init__(self, ...):
        ...
        if reservationid:
            ...
```

The new class is complete.

16. ❑ Below the `Part D` comment are six `name`*N* assignments. The six `flight`*N* objects created earlier will be also used.

    Use each `name`*N* and `flight`*N* to create a list of *`Reservation`* instances. The class constructor will assign `reservationid`.

    The data for each new `Reservation` instances:
    - For the `name` `Pat Holder`, `flightref` will be `flight1`
    - For the `name` `Peter Smith`, `flightref` will be `flight2`
    - For the `name` `Guy Gildersleeve`, `flightref` will be `flight3`
    - For the `name` `Janet Rider`, `flightref` will be `flight4`
    - For the `name` `Lynn Jasper`, `flightref` will be `flight5`
    - For the `name` `Ian Rouselle`, `flightref` will be `flight6`

17. ❑ Display the list's contents.

    Display the `reservationid`, `name`, and `flightnum` and `cost` for each new reservation.

Hint...

`flightnum` and `cost` are attributes within the `Flight` object, `flightref`.

Another hint...

Assuming this class:

```
class Reservation:
    def __init__(self, name=None, reservationid=None,
flightref=None):
        self.name = name
        if reservationid:
            self.reservationid = reservationid
        else:
            self.reservationid = reservation.next()
        self.flightref= flightref
```

With this instance:

```
res1 = Reservation(name=name1, flightref=flight1)
```

To access `flightnum`:

```
res1.flightref.flightnum
```

The new class has been tested.

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.

Objective

In this exercise, you will gain experience in taking advantage of module importing to use existing code by creating a module file for use in another program.

Preparing the module file and testing its name

This exercise will build on your work from Exercise 5.2. Some of the previous coding has been copied over to this project.

1. ❑ Open the `Ex6_1` project and the `airlineclasses.py` file. This file is based on the solution from Exercise 5.2, *excluding any bonus steps*.

   Within the file, review the:
   - Classes: `Trip`, `Flight`, and `Cruise`
   - Testing coding composed of
     - `Flight` and `Cruise` instance assignments
     - Attribute printing

2. ❑ Enclose the existing testing code within a function named `test_pgm()`.

Hint...

The testing code is the `Flight` and `Cruise` number assignments and printing. It should be below the `Part C` comment.

Another hint...

This coding must be indented as part of the `test_pgm()` function. For example:

```
class Trip:
...
class Cruise(Trip):
....
  def discount(self):
    ...

# Part C
def test_pgm():
  flight1 = Flight(flightnum=102, ...
  ...

if __name__ == '__main__' :
  test_pgm()
```

3. ❑ At the global level, add a conditional test to compare the module's `__name__` attribute to the string `'__main__'`. If `True`, then execute the testing code within `test_pgm()`.

   Run the program to verify that, when run as a program, the `test_pgm()` function is executed.

Hint...

To test, you may use:

```
if __name__ == '__main__':
```

Another hint...

Note the two underscores on either side of `name` and `main`:

`__name__` and `__main__`.

The module file is complete.

Using the newly created module from another program

4. ❑ The new module file is to be `imported` and used to create `Flight` objects.

   Close the `airlineclasses.py` editor pane.

5. ❑ Open the `Ex6_1.py` file. Make the following insertion above the `Part A` comment.

   Add the `import` statement to make the `airlineclasses` module available in this program.

Hint...

The name of the module does not include the `.py` extension.
Use `import as` for a shorter name.

6. ❑ Below `Part A` are comments containing an assignment to `flightlist`.

   Each list item should be a `Flight` object. The class name is missing.

   Uncomment this block of coding and edit to create `Flight` objects for the list.

Hint...

The assignment needs the qualified class name. For example:

```
flightlist = [
   modulename.classname(flightnum = 336, departcity = "HKG" ...

   modulename.classname(flightnum = 337, departcity = "HNL" ...

   modulename.classname(flightnum = 660, departcity = "CDG" ...

   ...
```

Another hint...

The `Flight` class is within the module `airlineclasses`.

Use `airlineclasses.Flight` to create the instances.

7. ❑ Below the `Part B` comment, use a loop to:
   - Display the `flghtnum` and `cost` attributes from each `Flight` object referenced in `flightlist`
   - Call the `discount()` method for each `Flight` object
   - Display the `flghtnum` and updated `cost` attributes

Moving from a list of instances to a dictionary of instances

8. ❑ Create an empty dictionary.

   Assign the list's `Flight` objects to that dictionary.

   The dictionary key will be the `flightnum` attribute. The associated value will be the entire `Flight` object.

Having `flightnum` as the key, and also contained in the value's `Flight` object, will cause no problems.

Hint...

Some of the coding may be similar to:

```
test_flight_dict = {}
for flt in flightlist:
  test_flight_dict[flt.flightnum] = flt
```

Another hint...

The empty dictionary must be created before a key–value assignment.

The dictionary is complete.

9. ❑ For this step, make the modifications below the `Part C` comment.

   Display the `flightnum` and `cost` from each dictionary value.

Congratulations! You now have a module of classes and methods that can be reused in other programs.

If you have more time: Using an additional module

10. ❑ Open the `reservationclass.py` file from the `Ex6_1` project. Review the file's contents.

The module contains:

- *The `nextid()` generator function*
- *The `reservation` iterable object*
- *The `Reservation` class definition, containing:*
  - *An `__init__` constructor method*
  - *The `name` attribute that will reference a passenger name*
  - *The `reservationid` attribute that was `yielded` from `nextid()`*
  - *The `flightref` attribute, which references a `Flight` object*

The generator function named `nextid()` was written for Exercise 4.2. It delivers successive unique values used for `reservationid`.

11. ❑ Close the `reservationclass.py` file.

12. ❑ Make further modifications in the `Ex6_1.py` file: At the top of the file, add a statement to make the `reservationclass` module available within your program.

Hint...

The style guide recommends putting each `import` statement on a separate line at the beginning of the source code.

Creating a `Reservation` instance

13. ❑ Below the `Part D` comment, there is a partial assignment to `tmpres1`. Uncomment this line and assign `tmpres1` a `Reservation` object using the arguments provided.

The `tmpres1` attribute `flightref` references a particular `Flight` object, in this example `flightlist[0]`.

Hint...

The class name must be qualified with the module name.

14. ❑ Display the `name`, `reservationid`, and `cost` values referenced by `tmpres1`.

Hint...

`name` and `reservationid` are attributes of the `Reservation` object.

`cost` is an attribute of the `flightref` object.

`flightref` is an attribute of `tmpres`.

Solution...

```
tmpres1.flightref.cost
```

15. ❑ The `tmpres1` attribute `flightref` reference was to `flightlist[0]`.

Uncomment the assignment of `tmpres2`. Modify this assignment to:

- Create a `Reservation` object
- Use the provided keyword arguments
- Use a `Flight` object from the dictionary created earlier; `336` should be a valid key

16. ❑ Display the `name`, `reservationid`, and `cost` values referenced from `tmpres2`.

An additional module has been added and used.

Using a standard library module

17. ❑ The `string` module contains string handling functions and constants. The constant `uppercase` is a string containing all the uppercase letters.

    Test all the `arrivecity` and `departcity` attributes from the dictionary of `Flight` objects.

    Display an error message if either of these attributes contains a character that is not an uppercase letter.

    Flight 681 `arrivecity` contains a digit.

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.

Objective

Handle various types of exceptions.

1. ❑ Open the `Ex7_1` project and the `Ex7_1.py` file.

2. ❑ Examine the `Ex7_1.py` file. This program calculates the Celsius equivalent of a Fahrenheit value.

The `print_ftoc()` function:

- *Loops through a list provided as a parameter*
- *Converts the text value into a floating point value*
- *Converts a Fahrenheit value to the Celsius equivalent*
- *Displays the calculated temperature*

Below the function are list assignments for the strings used in this exercise.

The final statement calls `print_ftoc()` using `temps1` as the argument.

3. ❑ Execute `Ex7_1.py`, and notice the Fahrenheit temperatures and calculated Celsius temperatures displayed.

4. ❑ Add a second call to `print_ftoc()` with `temps2` as the argument.

5. ❑ Run the program again. An exception will be raised.

You may need to scroll back though the console pane to see the error message.

Hint...

Notice the value at offset 2. The text value `'five'` cannot be converted to floating point.

The `ValueError` exception is raised.

6. ❑ Enclose both calls to `print_ftoc()` within a `try` statement. Add an `except` to handle the `ValueError` exception.

   If the `ValueError` exception is handled, display your own custom error message.

Hint...

A function call within a `try` statement will handle exceptions raised within that function.

Congratulations! You have handled an exception.

Exception instances

7. ❑ Notice the `except ValueError:` line.

   A `ValueError` instance provides the `args` attribute, a tuple passed to the exception class constructor method.

8. ❑ Modify the `except ValueError:` statement to create a reference to an instance of the class. Use the instance to display `args`.

The exception attribute has been used.

Nested `try`

The current coding construction:

```
try:
    print_ftoc(temps1)
    print_ftoc(temps2)
except ValueError:
```

causes execution to halt after the `ValueError` is handled.

Any remaining values from the lists are not processed.

9. ❑ The `float(temp)` function call within `print_ftoc()` causes the `ValueError` exception to be raised.

   Add a `try` statement within the `print_ftoc()` function. If a `ValueError` is raised:
   - Display an error message that the `ValueError` has been handled
   - Assign `0.0` to `temp` and complete the calculation

The innermost `try` caught the exception. Additional list values are now processed.

Raising an exception

10. ❑ Modify the main program to add a third call to `print_ftoc()` passing `temps3` as the argument.

    Add within the main program a new `except IndexError`. Display a descriptive error message if this exception is handled.

11. ❑ Modify the coding within `print_ftoc()`.

    If the length of its parameter is `0`, `raise` an `IndexError`.

An exception has been raised from within a function and handled within the main program.

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.

Objectives

In this exercise, you will learn to create data accessors from several types of files. To do this, you will:

- **Read data from text file in a CSV format**
- **Read class object data from `.pkl` files with `pickle`**
- **Read keyed data from `.dbm` files with `shelve`**

Introducing the `csv` module.

The Standard Library's `csv` module provides functions to process CSV files.

The `csv.reader()` function returns an iterator that delivers each row from a CSV file as a list of strings without quotation marks or newlines.

Syntax:

```
csv.reader(fileref)
```

For example:

```
import csv
with open('pathname', 'r') as fileref:
    reader = csv.reader(fileref)  # reader is the iterable object
    for oneline in reader:
        print oneline   # Display the list of strings returned from
reader
```

If the `'pathname'` file contained the line:

```
"ATL", "Atlanta/Hartsfield", 155.0
```

then `oneline` would reference the list of strings:

```
['ATL', 'Atlanta/Hartsfield', '155.0']
```

1. ❑ Open the `Ex7_2a` project and the `CSV_reader.py` file.

The file is an example using the `csv.reader()` function.

The second part of the program creates a dictionary from a line of CSV data:

- *The list of dictionary keys and the line of CSV data are passed to `zip()` to create the pairs of keys and values*
- *The pairs are passed to `dict()` to create a dictionary*

2. ❑ Execute `CSV_reader.py` to confirm its operation.

The lists of strings and dictionary were displayed.

Examining the exercise module files

Also within the `Ex7_2a` project, the `airlineclasses.py` file is provided. It contains the following classes:

- *`Trip`*
- *`Flight`*
- *`Reservation`*
- *`Aircraft`*
- *`Airport`*

The file also contains the `nextid()` generator function used to create the `reservationid` attribute for Reservation instances.

These classes and their attributes will be used. This module will be used throughout this exercise and many following exercises.

The `Reservation` class is used in later exercises.

The module is ready for import.

The `verifydicts.py` file is also provided. It contains the single function `verify_dicts()`.

Dictionaries of `Flight`, `Airport`, and `Aircraft` objects are created from various data stores. The `verify_dicts()` function is used to examine and verify the dictionaries created in this exercise and later exercises. The function will:

- *Display all dictionary lengths*
- *Display a single key, value pair from each dictionary*

Now that the data is coming from files, the dictionaries may be quite large. There will be over 3,000 `Flight` objects created.

A special note about the large number of `Flight` objects:

The `flightnum` attribute is no longer unique and cannot be used for the dictionary key. Flight number 336 may travel all seven days of the week.

The data stores have a unique integer that is used as the dictionary key for `Flight` objects. There is no change to the `Flight` class or the instances, only the dictionary key.

The module is also ready for `import`.

3. ❑ Open the `Ex7_2a.py` file.

- *The `csv` module is `imported`*
- *The `airlineclasses` module is `imported as ac`*
- *The `verifydicts` module is `imported as ve`*
- *The `sys` module is `imported`*

The main body of the program contains:

- *Three assignments of pathnames to the CSV files*
- *Within a `try` statement, three calls to the `get_any_dict()` function to assign the three dictionaries*
  - *`try` will handle any `IOError` exception*
- *A call to the `verify_dicts()` function to verify the three dictionaries*

There are three additional functions provided that handle converting a sequence into a `Flight`, `Airport`, or `Aircraft` object. The three functions are: `airport_mapper()`, `aircraft_mapper()`, and `flight_mapper()`.

Each mapper function:

- *Receives a sequence as an argument*
- *Creates the appropriate object type*
- *Maps the sequence elements into the proper object attributes*

The mapper functions return two values—a key to be used for a dictionary and the appropriate instance object.

Hint...

The *mapper* functions are unique to each class. They contain the mapping used to convert indexed sequence elements into instance attributes.

Another hint...

The *mapper* functions convert a flat sequence into a specific class instance with attributes and methods.

For example, a line from `airports.csv` file contains:

```
"HNL","Honolulu"
```

The `csv.reader()` iterator converts this line into a list of strings:

```
['HNL', 'Honolulu']
```

This list is passed as an argument to the `airport_mapper()` function. For example:

```
def airport_mapper(dataline):
```

The `airport_mapper()` function creates and assigns an `Airport` object. For example, `airlineclasses.py` contains:

```
class Airport(object):
  def __init__(self, citycode=None, city=None):
    self.citycode = citycode
    self.city = city
```

The `airport_mapper()` function maps its sequence into these specific attributes:

```
ap = ac.Airport(citycode = dataline[0],
                city = dataline[1])
```

The `airport_mapper()` function then returns a key, value pair:

```
return ap.citycode, ap
```

The `get_any_dict()` function handles reading a data file and assigning to a dictionary. Currently, the function:

- *Accepts a file name and a mapper function name as parameters*
- *Opens the file using `with` and `open()`*
- *Assigns a reader object, `linereader`, from the `csv.reader()` function*
- *Starts a loop to process the lines from the reader object by:*
  - *Displaying one line*
  - *Calling the `mapper()` function*
  - *Assigning the key, value pair returned from the `mapper()` function*
  - *Displaying the key, value pair*
  - *Terminating the loop after one iteration*
- *Returns the dictionary*

The `flight_mapper()` function returns `index`, a unique integer for the dictionary key. The `flightnum` attribute is not unique.

4. ❑ Execute the `Ex7_2a.py` file to view the results after processing one line from each data file.

5. ❑ Modify the `get_any_dict()` function:
   - Comment out the two `print` statements and the `break` statement
   - Use the key, value pair to assign into a dictionary
   - `return` the dictionary after the file has been processed

6. ❑ Execute the `Ex7_2a.py` file.

   The length of `city_code_dict` is `10`, the length of `aircraft_code_dict` is `4`, and the length of `flight_dict` is `3473`.

   CSV data files have been used to construct dictionaries of class instances.

   You may test the exception handling by changing one of the data file pathnames.

   An `IOError` exception is raised when the file is opened within `get_any_dict()`.

   The exception is handled in the main program.

Congratulations! You have used a Standard Library module to handle CSV files.

Reading pickle files

A separate project will be used for this section.

7. ❑ Open the `Ex7_2b` project.

   The `airlineclasses.py` and `verifydicts.py` module files are provided as before.

8. ❑ Open the `Ex7_2b.py` file.

9. ❑ Below the `Part A` comment, the modules are `imported`.

   The function header for `get_any_dict()` is provided. The function body is added in a later step in this exercise.

10. ❑ Below the `Part B` comment are assignments to pathnames for the existing `.pkl` files.

Each pickle file contains a dictionary of class instances.

For example, the `airports.pkl` file contains a dictionary of `Airport` class instances.

There are also three commented assignments for the dictionaries returned from `get_any_dict()` and a commented call to `verify_dicts()`.

11. ❑ Remove the # to uncomment these four lines.

The mapper functions are not included. No sequence to object mapping is needed with `pickle` data.

12. ❑ Add the coding within the main program to:
    - Call `get_any_dict()` from within a `try` statement
    - Handle an `IOError` raised within `get_any_dict()`

Hint...

To test this exception handling, change one of the pathnames to the files. The `IOError` exception is raised when attempting to `open()` the incorrect pathname.

13. ❑ Add the coding within the `get_any_dict()` function to:
    - Open the file name provided as a parameter
    - Read the pickle file and assign the contents to a dictionary
    - Close the pickle file
    - Return the dictionary

Hint...

The Course Notes have an example of using a pickle file to construct a dictionary.

The length of `city_code_dict` is `10`, the length of `aircraft_code_dict` is `4`, and the length of `flight_dict` is `3473`.

The new pickle data accessor has been added.

Creating the shelve data accessor

14. ❑ Open the `Ex7_2c` project and its `Ex7_2c.py` file for the final part of this exercise.

Below the `Part A` comment are the `import` statements, followed by the function header for `get_any_dict()`.

Below the `Part B` comment are assignments for the pathnames to the shelve files.
There are also three commented calls to `get_any_dict()` and a call to `verify_dicts()`.

15. ❑ Remove the # to uncomment these four lines.

Each shelve file contains a dictionary of class instances.

For example, the `airports.dbm` file contains a dictionary of `Airport` class instances.

16. ❑ Within the main program, handle any `IOError` that may occur from opening or reading the shelve file.

17. ❑ Within `get_any_dict()` add the coding to:
    - Create a dictionary from a shelve file
    - Return the dictionary

Assigning the entire dictionary would create a shared reference.

Hint...

See the Course Notes for an example of reading and writing a shelve file.

The length of `city_code_dict` is `10`, the length of `aircraft_code_dict` is `4`, the length of `flight_dict` is `3473`.

Dictionaries can now be constructed from several types of data sources.

18. ❑ Within the `Ex7_2c.py` file, create a new function to insert additional `Airport` objects into a shelve file.

    The function body should be **before** the main program within the source code. Add the function above the `# Part B` comment.

You may want to make a copy of the `C:\Course\1905\Data\airports.dbm` file before attempting to change it.

19. ❑ Create a function named `new_city()`:
    - It should accept two parameters:
      - A string to assign to the attribute `citycode`
      - A string to assign to the attribute `city`
    - Create and initialize an `Airport` object from the parameters
    - Update the shelve file; `citycode` is the key

20. ❑ Call the `new_city()` function twice using these tuples as the argument:

```
('LAX', 'Los Angeles')
('MSY', 'New Orleans')
```

21. ❑ Rebuild `city_code_dict` from the updated shelve file, then use `verify_dicts()` to check the updated shelve file contents.

`city_code_dict` should now have 12 elements.

22. ❑ Create a new function to remove `Airport` objects from a shelve file.

You may want to make a copy of the `C:\Course\1905\Data\airports.dbm` file before attempting to change it.

The `del` statement will remove an object. From the Python console, try:

```
city = 'Paris'
print city
del city
print city
```

23. ❑ Create a function named `del_city()` with these capabilites:
    - Accepts a parameter that is a dictionary key
    - Removes the dictionary item with that key from the shelve file

24. ❑ Add the statements to call the `del_city()` function twice:

```
del_city('LAX')
del_city('MSY')
```

25. ❑ Rebuild `city_code_dict`, then use `verify_dicts()` to check the updated shelve file contents.

`city_code_dict` should now reference 10 values.

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.

Objectives

You will write new data accessor functions to process data stored in a MySQL database. To do this, you will

- **Create data accessor functions for a MySQL relational database**
- **Execute SQL statements within the Python code**
- **Construct dictionaries from the values retrieved**

Creating the relational database data accessor functions

1. ❑ Open the `Ex8_1.py` project and `Ex8_1_Describe.py` file.

2. ❑ Review the components of the `Ex8_1_Describe.py` file:
   - Modules are imported
     - The `MySQLdb` module provides the database API
     - The `getpass` module provides the `getpass()` function used for password prompting
   - The `open_connection()` function connects and authenticates the database user and returns a connection object
   - The `describe_tables()` function generates and displays table information
     - The table name and its column names are displayed
     - The column names are collected into a list
     - A dictionary is created from the column names and a single retrieved row
     - The dictionary is displayed
   - The main program calls functions from within a `try` to:
     - Open the connection
     - Display table names, column names, and row dictionary
     - Close the connection

   The `open_connection()` function will prompt the user for the database password.

   To skip password prompting, comment out those lines within `open_connection()` and use the password assigned in the coding.

3. ❑ Execute `Ex_8_1_Describe.py` to verify the database can be queried with the password entered.

4. ❑ In the Eclipse Console pane near the bottom of the screen, you are prompted for the database password. Click in the console pane to enter text here. The password is **`ltree`**

   Experiment with an incorrect password to verify the exception handling for an `OperationalError` was used.

   When executing from Eclipse, the password is `echoed` and visible when using `getpass()`.

   When run from a command line, the password is not `echoed` and not visible when using `getpass()`.

Hint...

Ask your instructor for help, if needed, to understand the provided coding.

The database is available.

5. ❑ Open the `Ex8_1.py` file for this exercise. Review the statements below the `Part A` comment.

   The `MySQLdb` and `sys` modules are imported. The `getpass()` function is imported from the `getpass` module.

   The other two modules were used in the Chapter 7 exercises:
   - *`airlineclasses.py` is imported to provide the class definitions*
   - *`verifydicts.py` is imported to examine the dictionaries after their creation*

   The `airport_mapper()`, `aircraft_mapper()`, and `flight_mapper()` functions used earlier are provided to map sequences into objects and return `key, value` pairs

6. ❑ Notice the data accessor function, `get_any_dict()`, below the `Part B` comment.

The function will be very similar to the `get_any_dict()` function created in Exercise 7.1. Its two parameters are the database table name and the mapper function name for that table.

Add the statements in this function to:

- *Create a cursor object*
- *Execute a SQL statement to fetch the data from a database table*
- *Call an object mapper function*
- *Return a dictionary of objects constructed from the database*

Below the `Part C` comment are some additional comments providing the table names and associated mapper functions to be used. The same information is provided in the table below:

Class	Table Name	Mapper Function Name
`Airport`	`'airport'`	`airport_mapper()`
`Aircraft`	`'aircraft'`	`aircraft_mapper()`
`Flight`	`'flights'`	`flight_mapper()`
`Reservation`	`'reservations'`	`reservation_mapper()`

7. ❑ Within the block of statements monitored by `try`, immediately after the `connection` object has been assigned, add:
   - The three calls to `get_any_dict()` for `Airport`, `Aircraft`, and `Flight` data
   - The call to `verify_dicts()` to display samples from each dictionary

The length of `city_code_dict` is `10`, the length of `aircraft_code_dict` is `4`, and the length of `flight_dict` is `3473`.

Congratulations! You have written new data accessor functions to process data stored in a MySQL database.

If you have more time, extend the exception handling.

8. ❑ Continue editing the `Ex8_1.py` file.

   A `ProgrammingError` exception is raised if there is an error in the SQL.

9. ❑ Extend the `try` statement to handle a `ProgrammingError` exception.

   For testing purposes, you can raise this exception by using an incorrect table name.

Using the `Reservation` class

These bonus steps include accessing another database table containing `Reservation` information.

10. ❑ Open your `airlineclasses.py` file. Review the `Reservation` class definition.

The `Reservation` class contains three attributes:

- *`name`: a reference to a passenger's name*
- *`reservationid`: generated by the `nextid()` function when the reservation was created*
- *`flightref`, a reference to the `Flight` object*

The `reservation_mapper()` function provided in `Ex_8_1.py` unpacks a sequence and returns a `reservationid` and a `Reservation` object for assignment into a dictionary.

11. ❑ Add the necessary function call within the `try` statement to assign `reservation_dict` the reference returned by `get_any_dict()`.

12. ❑ Replace `import verifydicts as ve` with `import verify4dicts as ve`.

    The `verify4dicts.py` file and its `verifydicts()` function will display contents from `reservation_dict`, as well.

13. ❑ Add `reservation_dict` to the argument list for `verifydicts()`.

The `reservations` table has 10 rows. The first `reservationid` is `'200A'`, and the `name` is `'Bob Jones'`.

Adding a parameterized retrieval

Hint...

Examples of parameterized SQL statements are in the Course Notes.

14. ❑ Add statements to:
    - Request the user to enter a desired `reservationid`
    - Read in the value
    - Select and retrieve all rows from the `reservations` table where `reservationid` matches the input value
    - Display the `reservationid`, `name`, and `flightref` for any matching row

Hint...

The `reservationid`, `name`, and `flightref` are at offsets `[0]`, `[1]`, and `[2]` within each tuple referenced by the cursor.

`200A`, `201B`, and `202C` are all valid `reservationid` values.

Dereferencing a `Flight` object

The `flightref` attribute of a `Reservation` object is a key for the `flight_dict` dictionary.

15. ❑ Display the attributes of the `Flight` that was referenced from the `Reservation` object that was selected.

Creating a dictionary from database table column names

16. ❑ Review the `describe_tables()` function from `Ex8_1_Describe.py`. This function displays column names and creates a dictionary from the column names and a single retrieved row.

17. ❑ Modify the coding that displays the result of the parameterized query above.

    Create a simple dictionary where:
    - The dictionary key is the column name
    - The dictionary value is the contents of that field from the database

    Display this dictionary.

Hint...

The cursor provides data as a sequence.

Sequence pairs can be combined to create a dictionary.

Another hint...

The `zip()` and `dict()` functions may be used.

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.

Objectives

You will create an interactive graphical interface. To do this, you will

- **Display text in labels**
- **Create buttons with callback functions**
- **Display text with a `ScrolledText` widget**

Reviewing the existing coding

1. ❑ Open the `Ex9_1` project and the `Ex9_1a.py` file.

2. ❑ Execute the program and notice the components of the GUI. Compare the GUI to the the lines of source code that created and displayed the widgets.

3. ❑ There are widgets that hold other widgets:
   - The `Tk` root window with its title bar
   - The two `Frame` widgets with different relief styles

   Identify the widgets held within the two `Frames`:
   - A `ScrolledText` widget within the upper `Frame`
   - A `Label`, an `Entry`, and two `Buttons` within the lower `Frame`
   - The `Frames` and `ScrolledText` widgets are displayed using `pack()`
   - The other widgets are displayed in `column` and `row` locations using `grid()`

   The `Button` widgets have callback methods assigned to the `command` attribute, `display_f()`, and `display_c()`. These methods execute global functions to perform temperature conversion.

   The `Entry` widget will be used later in the exercise.
   This program will serve as a template for the GUI to be created. The GUI will be enhanced to:
   - *Display the converted temperatures in the `ScrolledText` widget*
   - *Accept a temperature value in the `Entry` widget and convert it to either Fahrenheit or Celsius, depending on the button used*

4. ❑ Click the buttons several times to confirm that the global functions display values in the Eclipse console pane.

Hint...

You may need to move the GUI to the side to see the console pane in Eclipse.

It is good program design to keep non-GUI calculations, like temperature conversion, outside of any GUI classes.

`50.0C` is equivalent to `122.0F`.

`50.0F` is equivalent to `10.0C`.

5. ❑ Close the GUI before modifying the source code.

Modify the label text

6. ❑ The `Label` widget currently shows `'Default prompt line'`.

   Assign to the `prompt` variable this string: `'Convert Celsius to Fahrenheit or Fahrenheit to Celsius'`

7. ❑ Execute the program again to confirm the new text in the `Label` widget.

   Close the GUI window before continuing modifications to the source code.

Writing to `ScrolledText`

8. ❑ Modify the global `ctof()` and `ftoc()` functions so that the original temperature and the converted temperature are `returned`.

9. ❑ Enhance the `display_c()` and `display_f()` methods so that the two values `returned` are appended to the `ScrolledText` widget.

   Test the program and test each button.

Hint...

The `insert()` method is used.

Use the string **`'end'`** as the first argument so additional `insert()` executions append within the `ScrolledText` widget.

Another hint...

Numeric values must be converted to text.

The original and converted temperatures are displayed in the `ScrolledText` widget.

Adding an additional button

10. ❑ Add an additional button to the right side of the lower frame of the GUI. This button should terminate the GUI.

    Create the button within the `setup_gui()` method so that its:
    - Parent widget is `bottomframe`
    - `text` attribute is assigned `'Exit'`
    - `command` attribute is the `quit` method

11. ❑ Use `grid()` to place the new button one column to the right of the `ftocbutton` widget. Test the program and all buttons.

Hint...

There are examples in the Course Notes and in the `Tk-Examples` project files.

Choose the appropriate `column` and `row` values for `grid()`.

Another hint...

The `grid()` method `row` parameter remains `0`; the `column` parameter is `5`.

Congratulations! You have created a graphical interface using `Label`, `Button`, and `ScrolledText` widgets.

If you have more time, investigate the `Entry` widget.

12. ❑ The `Entry` widget provided has remained unused so far.

Review the coding that creates the `Entry` widget named `self.input`:

- `self.temp` is an instance of class `StringVar`
- `self.temp` is assigned to the `textvariable` attribute of the `Entry`

`self.temp` will reference the *string* entered into the `Entry` widget name.

13. ❑ Add coding within the `display_f()` and `display_c()` methods to:

- Remove the assignment `base = 50.0`
- Assign to `base` the value entered into the `Entry` widget
- Continue to call `ctof()` or `ftoc()` passing `base` as the argument
- Clear out the `Entry` widget field

Hint...

Instances of class `StingVar` have a `get()` method to retrieve the string entered and a `set()` method to assign to the input field.

Another hint...

`self.temp.get()` returns the value entered into the `Entry` widget.

`self.temp.set('')` assigns an empty string to the input field, clearing any previous value.

14. ❑ The parameter for `ftoc()` and `ctof()` will now be a string.

    Add calls to `float()` to convert the string into a floating point value.

15. ❑ Test the program by entering these values and clicking the button for the desired conversion:
    `0`
    `100`
    `212`

`0.0C is 32.0F`
`0.0F is -17.7C`

`100.0C is 212.0F`
`100.0F is 37.8C`

`212C is 413.6F`
`212F is 100.0C`

16. ❑ Try this input string:
    `five`

`five` causes a `TypeError` exception to be raised.

17. ❑ Add a `try` statement within both conversion methods. If a `ValueError` exception is raised, perform the calculation using `0.0` as the value to convert.

Hint...

The `try` statement belongs in the functions that attempted the conversion, `ftoc()` and `ctof()`.

18. ❑ Try this input string once again:
`five`

The `TypeError` exception handled and the calculation proceeded, using `0.0` instead.

19. ❑ Add another button to the bottom frame that will clear the text from the `ScrolledText` widget.

Add a method to perform this action when the button is clicked.

This GUI is complete.

Attaching a GUI to the airline database

20. ❑ Open the `Ex9_1b.py` file to review its contents.

This program imports the `airlinedicts` module. This module file will:
- *Import the classes from the `airlineclasses` module*
- *Query the database and assign three dictionaries*
  - *`city_code_dict`, `aircraft_code_dict`, and `flight_dict`*

Similar work was done in Hands-On Exercise 8.1

21. ❑ Execute the file. The database login password is assigned in the source code.

Hint...

To restore prompting for the database password, uncomment the call to `getpass()` within `airlinedicts.py`.

Be aware that when you enter the password, Eclipse may hide the GUI from view. Minimize Eclipse or move it to the side after entering the password. The password is `ltree`

The `city_code_dict` key and a reference to an `Airport` object will display.

Radio buttons have been tested.

22. ❑ Modify the `showinfo()` method to:
    - Use the value from the `Radiobutton` selected to search within `flight_dict` for a matching *`departcity`*
    - Display within the `ScrolledText` widget some `Flight` attributes when the `departcity` matches the value from the `Radiobutton`
        - Display these attributes: `flightnum`, `departcity`, and `arrivecity`

The airline database has a GUI front end.

Creating a simple GUI for external executables

23. ❑ Open the `Ex9_1c.py` file within the `Ex9_1` project.

The file contains only comments about using the Python built-in function `execfile()`.

24. ❑ Follow the comments to create a GUI that uses `execfile()` to run additional Python programs.

Hint...

Only the four main parts of a Python `Tkinter` program are needed.

Any Python program can be launched from a GUI.

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.

Objectives
In this exercise, you will gain experience working with the Django web application framework. You will build a simple workflow that displays airline related information. The steps involved:

- **Write a view function to handle the request**
- **Write a model function to handle the data retrieval**
- **Configure the URL dispatcher to map your inbound URL to the view function**

Reviewing the view, model, and template components

1. ❑ In Eclipse, open the project folder `Ex10_1`.

There is a package in the folder named `Ex10_1`. This project is a Django project created using the PyDev plugin for Eclipse.

2. ❑ Open the `airline` package. The view, model, and other supporting programs are stored here.

   The `airlinedicts.py` file is provided. It:
   - Connects to the database
   - Retrieves all rows and constructs dictionaries of `Flight`, `Airport`, and `Aircraft` objects

   The `airlineclasses.py` file is provided. It contains the class definitions used in `airlinedicts.py`

3. ❑ Review the view component. Open the file `views.py` and review the coding.
   - The `airlinelookup` module is imported. This file contains the model functions for data retrieval.
   - A partial `get_city_code()` function is provided:
     - This function header is provided with two parameters:
       - The HTTP request data, named `request`
       - The URL-supplied airport code, named `citycode`
   - Some of the other coding provided:
     - `citycode` is converted to uppercase
     - An empty dictionary named `display_dict` is assigned. It will reference the values passed into the HTML template file
     - `return render_to_response` call with the HTML template file name and `display_dict` as arguments

This view function will be called from `urls.py`.

4. ❑ Review the model component. Open the `airlinelookup.py` file located in the `airline` project. Review the provided coding.
   - The `airlinedicts` module is imported
     - The `flight_dict`, `city_code_dict`, and `reservation_dict` dictionaries are created in this module
   - The local variable `city_code_dict` references `city_code_dict` created within `airlinedicts`
   - The `def find_city(citycode):` function header
   - The `pass` statement for the function's body

Hint...

You may also review the `airlineclasses.py` and `airlinedicts.py` files if desired. These were used in previous exercises.

Another hint...

The database password is assigned in the source code.

5. ❑ Review the template component.

   Open the `templates` folder within `Ex10_1`. Open the `airport_code.html` file.

   This file contains two variables named `citycode` and `city`. These are dictionary keys from the dictionary provided by the view.

Writing view and model functions to handle requests for country name

6. ❑ Complete the view component. Edit the `views.py` file.

   Modify the `get_city_code` function to call the `find_city()` function, the model function provided in the `airlinelookup` module, passing `citycode` as the argument.

7. ❑ `find_city()` will return the name of the corresponding `city` or a `'not found'` string.

   Assign two `key, value` pairs to `display_dict`:
   - The dictionary keys are the variable names to be used in `airport_code.html`, the strings `'citycode'` and `'city'`
   - The dictionary values are `citycode` and the reference returned from the `find_city()` model function assigned above

   The view portion of the application is completed.

8. ❑ Complete the model component. Edit the `airlinelookup.py` file.

   Remove or comment the existing `pass` statement.

9. ❑ Add statements to perform the following:
   - Test whether the `citycode` parameter is an actual key in the existing `city_code_dict` dictionary.
   - Return either:
     - The associated `Airport`'s `city` attribute, or
     - An error message string indicating that `citycode` is not found

Hint...

You may wish to save the two edited files and run them to verify there are no errors before moving on.

Another hint...

This is a Python Run.

The model is completed.

Mapping URL to view

10. ❑ Within the `urls.py` file, add the URL mapping for a regular expression and an associated view function.

The incoming request will contain `/airline/`*`sequence_of_non_digits/`*

The regular expression should match a *`sequence_of_non_digits`*:

- `\D` is a single non-digit
- `\D+` is one or more non-digits

The string matching the regular expression should be referenced by `citycode`, the argument to the view function.

Hint...

The generic syntax is `(?P<`*`variable`*`>`*`RegEx`*`)`.

Solution...

```
(?P<citycode>\D+)
```

The URL mapping completed.

Running the application

Before running, make sure there is no existing web server instance running from the command prompt and from Eclipse. If necessary, close the command prompt window and click the red box in the Eclipse console window to terminate the running process.

11. ❑ In the PyDev Package Explorer pane, select the project `Ex10_1` by clicking it once. Right-click and, from the resulting pop-up menu, select **Run As | PyDev: Django**.

Right-click on the project, not the package.

In the Eclipse console window, you should see a message stating that the web server application is starting.

Hint...

Development server is running at `http://127.0.0.1:8000/`

12. ❑ Open the Internet Explorer web browser and enter the URL **`http://localhost:8000/airline/HNL`**

Verify that your page is displaying `Honolulu`.

13. ❑ Try a few of the other airport codes, and ensure that your application is working as expected. Try some lowercase airport codes, as well, and some unknown airport codes.

These airport codes should be found:

- `NRT`
- `nrt`
- `Cdg`

These airport codes should not be found:

- `DFW`
- `MIA`

Congratulations! You have built a Django-powered web application.

If you have more time: Adding another application

14. ❑ Create a similar application to look up an aircraft `code` and display the associated `name`.

    As before, view and model functions must be written. The template is provided.

15. ❑ Within `views.py`, add the function `get_aircraft()` that will call the model, store the returned results in a dictionary, and assign to `display_dict` for the template.

    Add the `return render_to_response` call passing `display_dict` and the template file name for this request as arguments.

    The template file to use is `aircraft.html`.

16. ❑ Open the `airlineloopkup.py` file to add the new model function that tests whether the `code` parameter is an actual key in `aircraft_code_dict`.

    Return the associated value or an error message if it is not a key.


Hint...

`aircraft_code_dict` is assigned in the `airlinedicts.py` module.

You may use the qualified name or make a local reference.


Another hint...

`ad.aircraft_code_dict` is the qualified name.

17. ❑ Open the `urls.py` file and add an entry to map the URL to your view method.

    The incoming request will contain `/airline/digit/`

    An aircraft `code` character is a single digit. Construct the regular expression to match that pattern.

Hint...

`[0-9]` or `\d` are two regular expressions that match a single digit.

18. ❑ Stop and restart the web server before testing.

19. ❑ Open the Internet Explorer web browser and enter the URL **`http://localhost:8000/airline/1`**

Verify that your page is displaying `Canadian Regional Jet`.

20. ❑ Try a few of the other aircraft `codes` and ensure that your application is working as expected.

    The aircraft `codes` `1-4` should be found.

    Any other digit is not found.

Hint...

The single digit from the URL mapper is a character.

The dictionary uses an integer as the key.

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.