Exercise Manual for Course 1905

Python Programming Introduction

1905/MA/H.1/409/G.3

by Frank Schmidt

Technical Editor:
Alexander Lapajne

Exercise Manual Contents

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	**Stop**
?	**Question**	🏆	**Congratulations**
i	**Information**	BONUS	**Bonus**
🧩	**Solution/Answer**		

Hands-On Exercise 2.1:
Arithmetic and Numeric Types

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 PyCharm if it has not already been started.

Close any open editor windows.

From the PyCharm Projects window, open the `Ex2_1` folder. From there, open the `Ex2_1.py` file for this exercise.

The contents of the file are in the editor window.

2. ❑ Run this program.

You may execute the editor window's contents several ways.

For the first execution, right-click within the editor window and choose "Run" from the menu.

For subsequent executions you can:

- *Run as above*
- *You can use the Run menu and select the file*
- *You can use the Run button on the PyCharm task bar*

The standard output can be viewed in the Run window.

Any errors also appear in the Run window.

Look for the message `This is exercise 2.1` in the Run window 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 numbers and display the result of `num1 / num2`.

Execute the program.

Hint ...

The `int()` function will convert a string to an integer.

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

The string must contain values that can be converted.

You can assign the function return value for use in later calculations.

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

The result of the division is `0.5555555555555556` whether integer or floating point conversions are used.

We will learn how to format the output later in the course.

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

You now have a working arithmetic calculation.

Mixing types in arithmetic and precedence rules

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

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

There are string assignments to `paris_temp` and `honolulu_temp`. The variable `freezing` is assigned a floating point object.

The strings will need to be converted into a numeric type to perform the following calculations.

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.

5. ❑ 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

6. ❑ 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.

7. ❑ For this problem, you will need to `import` the `math` module.

Add your own assignments and calculations.

A circle has an area of 7.5 miles. What is the radius of this circle?

Hint...

The formula to calculate the area of a circle is `area =  pi * radius ** 2`

- The values of `area` and `pi` known
- The `sqrt()` function and `pi` are available through the `math` module

Congratulations! You have completed the bonus exercise.

STOP

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

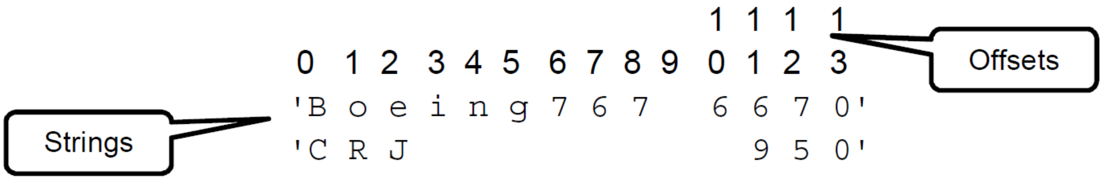
- **Use slicing techniques to extract substrings**
- **Use built-in methods to test strings**
- **Process a Comma-Separated Value (CSV) string**

Extracting string slicing and concatenation

1. ❑ Open the `Ex2_2` folder. 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.



```
                      1 1 1 1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3
'B o e i n g 7 6 7   6 6 7 0'
'C R J                 9 5 0'
```

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 `planeN` string.

Hint...

Use string slicing to extract type and range for each `planeN`.

Another hint...

An unbounded slice terminates at the end of the string.

Hands-On Exercise 2.2: Strings and `if` (continued)

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.

Using string methods

4. □ 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`

5. ❑ 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

6. ❑ 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

7. ❑ 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.

8. ❑ 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.

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

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

10. ❑ 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 precedes the offset of the `","` comma.

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

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

    Using a loop and a function would help with repetitive steps. Loops are discussed in Chapter 3, functions are discussed in Chapter 4.

12. ❏ 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.

STOP

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` folder 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` is assigned. This list contains details of an airline flight.

   Comments describe the mapping of the list values to the details of an airline flight.

   `[0]` is the `departcity`
   `[1]` is the `arrivecity`
   `[2]` is the `departdaytime`
   `[3]` is the `arrivedaytime`
   `[4]` is the `cost`
   `[5]` is the `code`

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

Hint...

Sequence unpacking will require enough variable names, at least four.

The wildcard notation can be used to assign the final four objects.

Applying list methods

Many list methods change the list in place and return `None`.

A common mistake is to reassign the return value back to the list variable.

Do this: `data.sort()`

Not this: `data = data.sort()`

4. ❑ Reverse the contents 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.

Assignment creates a shared reference.

Congratulations! You have managed lists and slices of lists.

If you have more time, explore more list handling.

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.

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.

Additional list modification methods

13. ❑ Extend `codelist` by placing `'ABC'` before the first element of the list.

14. ❑ Extend `codelist` by placing `'XYZ'` after the last element of the list.

Display the new contents of `codelist`.

Hint...

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

Comparing lists and tuples

15. ❑ Create a tuple from `codelist` and assign it to `codetupe`.

16. ❑ Test whether the length of `codelist` remains equal with the length of `codetupe`.

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

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

These will fail. Tuples are an immutable type.

Additional list handling methods

18. ❑ Look up the `sorted()` function in the Python documentation.

Discover the purpose, return value, and reverse parameter for this function.

Hint...

The PyDoc documentation browser is available from the Windows Start button, under the Python 3.10 folder.

Or, from PyCharm Python console, you can use the `help()` function.

19. ❏ Using `sorted()`, sort `codetupe` in reverse order and convert the value returned into a tuple.

Display this new tuple.

Congratulations! You have completed the bonus exercise.

STOP

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.

   Below the dictionary is a `for` loop that displays the keys from `city_code_dict`.

Hint...

> You can use the dictionary name without a method for access to the keys, instead of the `keys()` method.

2. ❑ Within the `print()` function, use the key to also display the corresponding value.

Membership testing using loops and `if`

3. ❑ 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.

4. ❑ 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

5. ❑ 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.

6. ❑ 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`

7. ❏ Below the `Part B` comment, the variable `flightlist` was 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 `departdaytime`*
- *`[3]` is the `arrivedaytime`*
- *`[4]` is the `cost`*
- *`[5]` is the `code`*

8. ❏ Use a list comprehension.

Create and display a list of the flight number for the flights that depart from `'HNL'`.

Hint...

The departure city is `[0]` of each list within `flightdict`.

✓ *The list of flight numbers from HNL is `[102, 132, 1572]`.*

9. ❑ Use a list comprehension.

Display the list of round trip flights, those that depart and arrive in the same airport.

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

10. ❑ 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 or dictionary name alone provides a view object of the dictionary keys. This can be converted into a list to enable sorting.

11. ❑ Below the `Part D` comments, `airports` is assigned a tuple of CSV strings. The first field is the airport code; the second is the city name.

Create and display a list of all airport codes.

Create and display a list of all city names.

Hint...

The `split()` function and `join()` function may help.

Congratulations! You have completed the bonus exercise.

STOP

This is the end of the exercise.

Hands-On Exercise 4.1: Creating and Calling Functions

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` folder. 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 airport name.

   `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 `departdaytime`
   `[3]` is the `arrivedaytime`
   `[4]` is the `cost`
   `[5]` is the `code`

2. ❑ Create a function named `list_all_cities()` that displays the three-letter airport code and the corresponding city 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. 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

4. □ 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

5. ❑ 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.

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

The function is now complete.

7. ❑ 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.

8. ❑ 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.

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

Hint...

Each dictionary key is the flight number.

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.

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

This function is now complete.

11. ❑ 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

12. ❑ Add the following two assignments:

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

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

14. ❑ 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.*

15. ❑ 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 subtracted. 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` function.

Hint...

The function body will contain only the calculation.

It could be:

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

Function calling a function

16. ❑ 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.

STOP

This is the end of the exercise.

Hands-On Exercise 5.1:
Classes and Initialization

Objectives

In this exercise, you will gain experience with classes that define data used for an airline.

- **Define attributes**
- **Add a new method**

Reviewing classes and the `__init__()` method

1. ❑ Open the `Ex5_1` folder and the `Ex5_1.py` file for this exercise. Notice the:
   - Class named `Trip`
   - `Class` attributes `cariblist` and `hawaiilist`
   - Constructor method with:
     - Keyword parameters and defaults
     - Assignments of the four attributes

   ✓ *The first class has been created.*

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

   There is a commented assignment to `mytrip` that supply the entire argument list keyword style. Uncomment these lines to create the instance.

3. ❑ Add the statements to create the instance using the data provided.

   Execute the program.

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

4. ❑ Add `print()` statements to display the instance attributes and also the two class attributes, `hawaii_list` and `carib_list`.

Hint...

Use the class name to access these class variables.

Solution...

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

Adding a method

5. ❑ 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`.

6. ❑ Using the `mytrip` instance created earlier, determine and display whether that was a round-trip.

Add an `if` statement to test the return value from `is_roundtrip()` and display the results.

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

7. ❑ Modify the `mytrip` instance assignment so that both `departcity` and `arrivecity` are the same value.

Execute the `is_roundtrip()` method again to verify a round trip is found.

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 attributes, and an additional method.

If you have more time, add more 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, three additional methods will be added to the `Trip` class. These methods all return Boolean values and are similar to the `is_roundtrip()`.

Add the following methods within the `Trip` class definition:

- `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 as assignment to `alltrips`, a list of `Trip` objects. There is also a commented function, `print_trip()`.

   Uncomment the assignments to construct the list of `Trip` instances.

   Uncomment the function `print_trip()`.

   *The * indicates each list is passed as positional arguments to the `Trip` constructor.*

10. ❑ Construct a loop to process all the elements of `alltrips` to:
    - Call `print_trip()` to display its attributes
    - Call the methods and display results from:
      - – `is_round_trip()`
      - – `is_caribbean()`
      - – `is_hawaiian()`
      - – `is_interisland()`

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

Additional classes

12. ❑ Class definitions should be toward the beginning of a source code file. Be sure to create them before or after the `Trip` class, but not indented within `Trip`.

    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`

13. ❑ 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.

14. ❑ 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` folder 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`
   - Four methods: `is_round_trip()`, `is_caribbean()`, `is_hawaiian()`, `is_interisland()`.

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

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

Modify the `Flight` class, make it a subclass of `Trip`.

The new class contains all the `Trip` attributes and has three additional attributes: `flightnum`, `cost`, and `code`.

The `Flight` constructor receives the `Trip` constructor arguments in `*args` and `**kwargs`

Within the `Flight` constructor, add three statements:

- Two assignments for the attributes `cost` and `code`
- Add a call to `super()` to call the `Trip` class constructor passing `*args` and `**kwargs` as the arguments

Hint...

The `Flight.__init__()` method parameter list contains `Flight` attributes `flightnum`, `cost`, and `code`.

The method also receives `Trip` attributes in `*args` and `**kwargs`.

Another hint...

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

The new subclass has been created.

Creating an instance and inheriting attributes

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

Uncomment the assignment to `trip`.

Display the attributes by using **`vars(mytrip)`** *`vars()` returns a dictionary of an object's attributes.*

5. ❑ Continue below the `Part C` comment, more test data has been provided within comments. There is a list of `Flight` objects

Uncomment the assignments to `allflights`. Note, we are passing arguments to both `Flight` and `Trip` constructors positionally.

Use this list in a loop to display the attributes of the individual `Flight` objects.

The new subclass has been tested.

Congratulations! You have added a subclass that contains additional attributes. You have created multiple objects from this new subclass.

If you have more time, try adding more methods.

6. ❑ 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.

7. ❏ Modify the loop that displays each `Flight` object:
   - Display the `flightnum` and original `cost`
   - Call the `discount()` method
   - Display the `flightnum` and new `cost`

✓ *Flights 102, 336, and 660 were discounted.*

Flights 317 and 204 were not discounted.

Congratulations! You have added a subclass that contains additional attributes and methods. You have created multiple objects from this new subclass and applied methods.

Creating a new class

One attribute will be an instance of another class.

8. ❏ Create a new class named `Reservation`. This will be a standalone class and does not inherit from `Trip` or `Flight`.

   The new class should be added near the top of the source code. Place it under the `Flight` class.

   Add the constructor for `Reservation`, it should accept three parameters and assign them to three attributes:
   - `name` will contain a passenger's name
   - `reservationid` will contain a unique string to identify this reservation
   - `flightref` will contain a `Flight` object

Hint...

Use any of the six `Flight` objects created earlier, `flight1`, `flight2`, etc.

Assign a single `Flight` to each value for `flightref`.

9. ❑ Below the `Part D` comment is `reservations`, a list of six dictionaries. Each dictionary will be used to construct a `Reservation` instance.

   The `flightref` key for each dictionary references a `None` value.

   Modify the dictionaries and assign a `Flight` instance for each `flightref` key.

10. ❑ Use the dictionaries in the `reservation` list to create the six new `Reservation` instances.

    Be sure to use `**` in the argument call so each dictionary is passed as keyword arguments.

11. ❑ 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
        self.reservationid = reservationid
        self.flightref= flightref
```

To access `flightnum`:

```
reservation.flightref.flightnum
```

✓ *The new class has been tested.*

Congratulations! You have completed the bonus exercise.

STOP

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` folder 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 `Trip`, and `Flight` classes.

   Near the end of the file, notice the test of __name__ and the conditional creation of `Trip` and `Flight` objects.

   You can read more about `vars()` in the Python documentation

   This dictionary named `data` provides the keyword arguments when calling the constructors.

   The module file is ready for testing and importing.

2. □ Execute `airlineclasses.py` as a standalone program to verify it works.

   The classes are ready to use for importing.

Using the newly created module from another program

3. □ Open the `Ex6_1.py` file. Near the top of the file, add the `import` statement to make the `airlineclasses` module available in this program.

Hands-On Exercise 6.1: Modules (continued)

Hint...

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

Use `from` *`module`* `import` *`object`* if you want to use unqualified names.

4. ❑ Execute `Ex6_1.py` to verify the `import` works and that `main_pgm()` is executed.

5. ❑ Add additional statements within `main_pgm()`.
   - Create a dictionary from `flight_attributes` and `flight_data`
   - Create a `Flight` object by passing the dictionary to the `Flight` class constructor
   - Display the attributes of the `Flight` object using `vars()`

Hint...

See `airlinedata.py` for an example using `dict()` and `zip()` with two sequences.

6. ❑ Execute `Ex6_1.py` to verify success.

Congratulations! You now have used classes from one module to build objects in another.

If you have more time, use an additional module.

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

The module contains:

- *The `Reservation` class definition, containing:*
  - *An `__init__` constructor method*
  - *The name attribute that will reference a passenger name*
  - *The `reservationid` attribute*
  - *The `flightref` attribute, which references a `Flight` object*

8. ❑ The `reservationdata.py` file is provided. It contains the data to create two `Reservation` objects.

   Review the assignments in this file.

Creating a `Reservation` instance

9. ❑ Extend your previous solution in `Ex6_1.py` to:
   - `import` the `reservationclass` and `reservationdata` modules
   - Create two `Reservation` objects using the tuples from `reservationdata.py`
   - Display the attributes of the `Reservation` objects, including the attributes of `flightref`, the embedded `Flight` object

✓ *Two additional modules have been added and used.*

Hint...

You will have two levels of attribute resolution to access `flightnum` and `cost`. For example:

`reservation``.flightres.flightnum`

✓ *The `Reservation` class has been tested.*

Using more modules from the Standard Library

10. ❏ Look up the `shutil` module in the Standard Library documentation.

Read about the `copy()` function in that module.

Use this function to make a copy of `reservationdata.py`. Name the new copy `reservationdata.backup`.

11. ❏ Look up the `glob` module in the Standard Library documentation.

Read about the `glob()` function in that module.

Use this function to display all filenames in the current directory that end with `.py`.

✓ *Additional Standard Library modules have been researched and used*

Congratulations! You have completed the bonus exercise.

STOP

This is the end of the exercise.

Objective

Handle various types of exceptions.

1. ❑ Open the `Ex7_1` folder 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 window 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.

7. □ Error messages should be sent to the proper file, `sys.stderr`.

   At the top of your file, add the statement to `import` the `sys` module.

   For the `print()` calls within the exception handler, send the error message to `sys.stderr`.

   In PyCharm's Run window, `sys.stderr` is in red.

Congratulations! You have handled an exception.

Exception instances

8. □ Notice the `except ValueError:` line.

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

9. □ 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.

10. ❑ 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

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

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

12. ❑ 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.

STOP

This is the end of the exercise.

Hands-On Exercise 7.2: Managing Files

Objectives

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

- **Read data from a text file in a CSV format**
- **Write data to a text file in a CSV format**
- **Create objects from text data files**

1. ❑ Open the `Ex7_2a` folder and the `Ex7_2a.py` file.

The exception handler is written to monitor exceptions within the `main()` function.

Within `main()` the variable `file` is assigned a pathname in a string. This is the input data file for this exercise.

Reading a text file

If you want to review the data file, you can open it with Notepad.

2. ❑ Add the statements within `main()` to:
    - Open the file for reading
    - Read each line from the file
    - Display each line after it is read
    - Close the file

Each line is a single string. Over 3000 strings are displayed.

The file can be opened, processed, and closed.

3. ❑ Extend your previous solution.

Convert each line of input from the file into a list of strings. Display these lists.

Hint...

The `split()` function may be helpful.

Each line is a list of strings. Over 3000 lists are displayed.

The strings were converted into sequences.

4. ❏ Each list describes one flight. The list offsets correspond to these values:
   - `flight_number, depart_city, arrive_city, depart_daytime, arrive_daytime, cost, code`

   The variable `search_flight` is assigned a `flight_number` as a string.

   Extend your previous solution:
   - Comment out the printing of every line
   - Test if a `flight_number` matches `search_flight`. If they match:
     – Display only the matching list

Writing a text file.

5. ❏ Extend your previous solution.   Add additional statements to write the matching lists to a new text file named "`search_flights.csv`". This file will be in the same folder as `Ex7_2.py`.

   Add the statements to open this file for writing.

   Write each list as a single CSV string.   You may need to insert the commas within each string.

Hint...

The `format()` function may be helpful, but the `join()` function may be easier.

6. ❏ Verify the contents of the new output file. Open the file using a text editor or open the file with PyCharm.

A new CSV file has been created.

Congratulations! You have read, written, and processed CSV files.

The CSV module

Introducing the csv module.

Using quotation marks around strings is common for CSV data.

Examine the `search_flight.csv` file created in the prior steps. All the values are strings, some strings contain `"`.

```
1587,"CUR","HNL","2022-01-02 12:00","2022-01-02
20:00",299.99,2
```

The Standard Library has tools to more easily manage this type of data.

Open the documentation for the Standard Library's `csv` module. Read the paragraph about the `Dictreader()` class.

The `csv.Dictreader()` function returns an iterator that delivers each row from a CSV file as a dictionary.

The dictionary keys come from the first line of the file.

The dictionary values are strings without quotation marks or newlines.

7. ❑ Open the `CSV_reader.py` file and review the use of the `csv.Dictreader()` function. Execute the file to see the results.

Within the `for` loop, notice:

- `Dictreader()` returns a dictionary
- The dictionary is used to construct an `Aircraft` object
- The `vars()` function returns the attribute names and values of the `Aircraft` object

✓ *Each line of the CSV file can create a new `Aircraft` instance.*

8. ❑ Open the `Ex7_2b` folder and the `Ex7_2b.py` file.

The `airlineclasses` and `csv` modules have been imported.

The exception handler is written to monitor exceptions within the `main()` function.

Within `main()` the variable `file` is assigned a pathname in a string. This is the input data file for this exercise.

You may wish to review the class descriptions for `Trip` and `Flight` within the `airlineclasses.py` file.

`C:\Course\1905\Data\flights.csv` file will be used. It has a header line that provides the dictionary keys for `csv.Dictreader`.

These dictionary keys are the same names as the attributes of `Trip` and `Flight` objects. The constructors can be called using keyword arguments.

Creating a list of `Flight` objects

9. ❑ Using `CSV_reader.py` as an example, add the statements within `main()` to:
   - Open the file for reading
   - Create a dictionary from each input line
   - Create a `Flight` object from the dictionary
   - Add the `Flight` object to a list

10. ❑ Display the number of `Flight` objects in the list.

Display the attributes of the final `Flight` added to the list.

✓ *Each line of the CSV file was used create a new `Flight` instance.*

Congratulations! You have completed the bonus exercise.

STOP

This is the end of the exercise.

Hands-On Exercise 8.1:
Accessing a Database

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

- **Create data accessor functions for a database**
- **Execute SQL statements within the Python code**

Creating the relational database data accessor functions

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

2. ❑ Review the components of the `Ex8_1_Describe.py` file:
    - Modules are imported
        - The `sqlite3` module provides the database API
    - The `open_connection()` function connects and returns a connection object
    - The `describe_tables()` function displays table information
        - The table name and its column names are displayed
        - The `CREATE TABLE` statement used is displayed
    - The main program calls functions from within a `try` to:
        - Open the connection
        - Display table names and column names
        - Close the connection

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

    Make note of the table names.

    ✓ *The database is available.*

4. ❑ Open the `Ex8_1.py` file for this exercise. Review the existing statements.

The required module is imported.

The `open_connection()` function returns a connection object for the `airline` database.

The exception handler will:

- *Call `open_connection()` and assign the connection object*
- *Call the `search_db()` function*
- *Handle some `sqlite3` exceptions*
- *Close the connection*

5. ❑ Modify the `search_db()` to:
   - Create a cursor object
   - Use the cursor to execute the SQL statement `"SELECT * FROM flights"`
   - Display the rows from the cursor

The `flights` table has been queried and displayed.

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

If you have more time, extend the database handling.

Creating a custom query

User input will be accepted to create a customized query.

6. □ Extend the previous solution to retrieve only certain flights from the `flights` table.

The SQL statement `"SELECT * FROM flights"` will be modified.

Modify the `search_db()` function to:

- Query the user for a flight number to search
- Create a parameterized SQL query to retrieve the rows where `flightnum` matches the input value
- Display the matching rows

Test the solution using flight number `1347`.

Hint...

When executing the parameterized SQL statement, be sure the query parameter is in a sequence. A tuple is recommended.

There were 7 occurrences of flight `1347`.

7. □ Execute the previous solution to retrieve only flight number `99999`.

That flight does not exist in the database, the `SELECT` statement returns no rows.

Inserting a row

The `new_flight` list has been assigned values that can be added to the `flights` database.

8. □ Extend the previous solution to insert a row into the `flights` table.

Create a new function for this work.

Within this new function:

- Create the parameterized SQL statement to `INSERT` the row
- Execute the connection's `commit()` method after the insertion

✓ *A new row for flight 99999 has been added.*

9. ❏ Extend the previous solution to remove a row from the `flights` table.

Create a new function for this work.

Within this new function:

- Create the parameterized SQL statement to `DELETE` the row inserted in the step above
- Execute the connection's `commit()` method after the deletion

✓ *The database has been queried and updated.*

Congratulations! You have completed the bonus exercise.

STOP

This is the end of the exercise.