# **Programming with Python**

# Module 5:



* Lists
* Dictionaries
* Structured Error Handling
* Functions
* Script Templates

## Lists

Lists are a simple way to hold a collection of objects. They are similar to tuples, but have a lot more flexibility and features.

“The list data type has some more methods. Here are all of the methods of list objects:”

list.**append**(x)

Add an item to the end of the list; equivalent to a[len(a):] = [x].

list.**extend**(L)

Extend the list by appending all the items in the given list; equivalent to a[len(a):] = L.

list.**insert**(i, x)

Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).

list.**remove**(x)

Remove the first item from the list whose value is x. It is an error if there is no such item.

list.**pop**([i])

Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

list.**index**(x)

Return the index in the list of the first item whose value is x. It is an error if there is no such item.

list.**count**(x)

Return the number of times x appears in the list.

list.**sort**()

Sort the items of the list, in place.

list.**reverse**()

Reverse the elements of the list, in place. " (Python Documentation, <https://docs.python.org/2/tutorial/datastructures.html#more-on-lists>, 2015)

Here is an example that uses some of the list methods:

#-------------------------------------------------#

# Title: Working with Lists

# Dev: RRoot

# Date: Jul 25, 2015

# ChangeLog: (Who, When, What)

# none yet

#-------------------------------------------------#

#1) A list and a tuple are very similar

lstRow1 = **[**"1","Bob Smith", "BSmith@Hotmail.com"**]**

tplRow1 = **("**1","Bob Smith", "BSmith@Hotmail.com"**)**

print(lstRow1)

print(tplRow1)

#2) Just like tuples your list can be nested (multi-dimensional)

print("\n--- multi-dimensional")

lstRow2 = ["2","Sue Jones", "SueJ@Yahoo.com"]

**lstTable** = [lstRow1, lstRow2]

print(lstTable)

#3) Lists have a number extra functions and properties

#3a) You can Append to a list without it recreating a new one

print("\n--- append")

lstRow1.**append**("555-1234")

print(lstTable)

# Tuples do not have these extras!

# AttributeError: 'tuple' object has no attribute 'append'

# tplData.append("555-1234")

#3b) You can Remove a item

print("\n--- remove")

lstRow1.**remove**("555-1234")

print(lstTable)

#3c) You can Insert data into a given spot

print("\n--- insert")

lstRow3 = ["3", "Joe James", "JoeJames@Gmail.com"]

lstTable.**insert**(0, lstRow3)

print(lstTable)

#3d) You can Sort the data

print("\n--- sort")

lstTable.**sort**()

print(lstTable)



**LAB 5-1: Working with Lists**

1. Create an application that uses a list to hold the following data:

|  |  |  |
| --- | --- | --- |
| Id | Name | Email |
| 1 | Bob Smith | BSmith@Hotmail.com |
| 2 | Sue Jones | SueJ@Yahoo.com |
| 3 | Joe James | JoeJames@Gmail.com |

1. Add code that lets users appends a new row of data.
2. Add a loop that lets the user keep adding rows.
3. Ask the user if they want to save the data to a file when they exit the loop.
4. Save the data to a file if they say 'yes'



## Dictionaries

### From the Python help files:

"Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys.” (Python Documentation, <https://docs.python.org/2/tutorial/datastructures.html#dictionaries>, 2015)

Here is an example:

#-------------------------------------------------#

# Title: Working with Lists

# Dev: RRoot

# Date: Jul 25, 2015

# ChangeLog: (Who, When, What)

# none yet

#-------------------------------------------------#

#1) A list and a dictionaries are very similar

# but dictionaries use "keys" instead of indexes

lstRow1 = ["1","Bob Smith", "BSmith@Hotmail.com"]

dicRow1 = {"ID":1,"Name":"Bob Smith", "Email":"BSmith@Hotmail.com"}

print(lstRow1)

print(dicRow1)

print(lstRow1[1])

print(dicRow1["Name"])

#2) dictionaries should can be nested (multi-dimensional),

# but may also be placed in a list to keep things simple.

print("\n--- multi-dimensional list")

dicRow2 = {"ID":"2","Name":"Sue Jones", "Email":"SueJ@Yahoo.com"}

lstTable = [dicRow1, dicRow2]

print(lstTable)

#3) You can loop through the list to see you dictionary objects

print("\n--- items in the list 'Table'")

for objRow in lstTable:

print(objRow)

#4) Dictionaries have a number extra functions and properties

#4a) items()

print("\n--- items()")

for myKey, myValue in dicRow1.items():

print(myKey, " = ", myValue )

#4b) values()

print("\n--- values()")

print(dicRow1.values())

#4b) keys()

print("\n--- keys()")

print(dicRow1.keys())



**LAB 5-2: Working with Dictionaries**

1. Create an application that uses a dictionary to hold the following data:

|  |  |  |
| --- | --- | --- |
| Id | Name | Email |
| 1 | Bob Smith | BSmith@Hotmail.com |
| 2 | Sue Jones | SueJ@Yahoo.com |
| 3 | Joe James | JoeJames@Gmail.com |

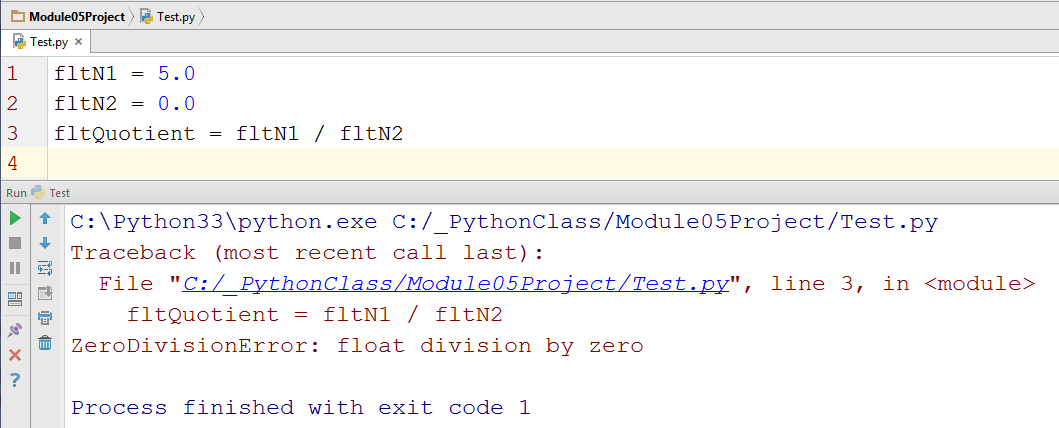
1. Add code that lets users appends a new row of data.
2. Add a loop that lets the user keep adding rows.
3. Ask the user if they want to save the data to a file when they exit the loop.
4. Save the data to a file if they say 'yes'



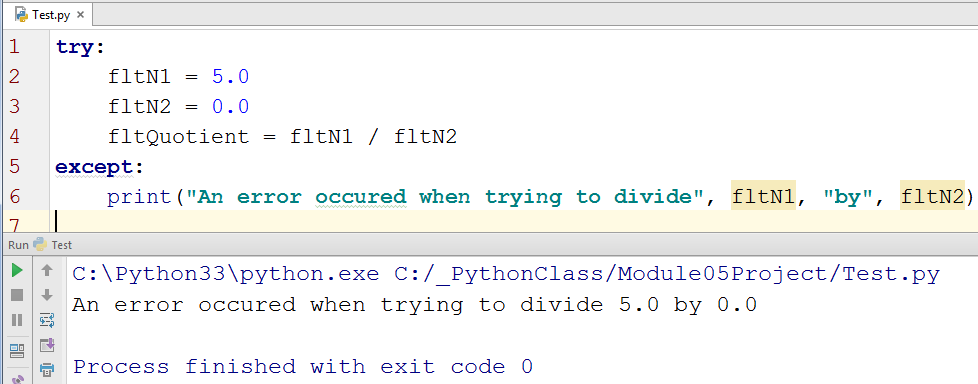
## Try-Except

You can trap errors in your programs using a try-except construct. One advantage of this is that provides a simple, organized way of grouping one of more statements to be processed. Another is that if an error occurs in those statements, Python will automatically move to another set of statements where you may handle the error in your own way (instead of the way python would normally do so).

For example, here is what a normal divide by zero error looks like.



While it works just fine for developers, it may be more complex than you would like. If that is the case, then you can add a Try-Except block to your code and customize the error as shown here:



## Functions

Functions are another way of grouping programming statements together. These statement can then be run later in your program by “calling” the function.

For example, let’s put our division code into a new function called DivideValues(). When the script starts it will load the function into memory, but it will wait to run its statements. In this following example the code will actually start running the first statement after the definition of the function and then move on to the next one, and the next. However on the next line it finds a call to the function and so jumps to the section of code where the function is defined. It runs all of the code inside the function and then returns to the finish the line of code where the function was called. After that, it proceeds to the next line (I have numbered these steps with comments).

def **DivideValues**(): #4  
 return (fltN1 / fltN2) #5  
  
fltN1 = float(input("Enter the first number: ")) #1  
fltN2 = float(input("Enter the second number: ")) #2  
print(**DivideValues**()) #3

print("Done") #6

## Script Templates

To make you scripts more professional you need consistency. On easy way to do this is by starting each script with a template.

Most programs can be divided into three different sections; Data, Processing, and Presentation (or Input-Output).

Because of this, often you can divide your code into to these sections. For instance you could start a new script by adding comment to it as shown here.

#-- Data --#

# declare variables and constants

#-- Processing --#

# perform tasks

#-- Presentation (Input/Output) --#

# get user input

# send program output

After that you could add code the different sections as follows.

#-- Data --#  
# declare variables and constants  
fltN1 = 0.0  
fltN2 = 0.0  
  
#-- Processing --#  
# perform tasks  
**def** DivideValues():  
 **return** (fltN1 / fltN2)  
  
#-- Presentation (Input/Output) --#  
# get user input  
fltN1 = float(input(**"Enter the first number: "**))  
fltN2 = float(input(**"Enter the second number: "**))  
# send program output  
print(DivideValues())

### Adding a Preface to scripts

When working as a developer you may also be asked to include a preface in all of your scripts. This is considered Meta data that describes things about the program.

#-------------------------------------------------#

# Title: <Type the name of the script here>

# Dev: <Type your name here>

# Date: <Type the day this script was first created>

# Desc: <Type a description of the script>

# ChangeLog: (Who, When, What)

# <Example: RRoot, 01/04/2012, Added more code>

#-------------------------------------------------#

#-- Data --#

# declare variables and constants

#-- Processing --#

# perform tasks

#-- Presentation (I/O) --#

# get user input

# send program output

## Review Chapter 5

“Specifically in this chapter, you’ll learn to do the following:

* Create, index, and slice a list
* Add and delete elements from a list
* Use list methods to append and sort a list
* Use nested sequences to represent even more complex information
* Use dictionaries to work with pairs of data
* Add and delete dictionary items”

(Chapter 5, Python Programming for the Absolute Beginner, Third Edition, [Dawson](http://library.books24x7.com.ezproxy.kcls.org/SearchResults.aspx?qdom=author&scol=%7ball%7d&qstr=Michael%20Dawson))