Introduction to Python Programming

Exercise Manual

This page intentionally left blank.

Contents

Exercise 1.1: Introduction to ipython

Startup

1. Log in as directed by the instructor.

2. Open a terminal window and execute the following:
   a. `cd ~/Introduction.to.Python`
   b. This is the home directory for the course.

3. Execute: `ls`
   a. You should see something similar to:

```
(base) student@ab-python3:~/Introduction.to.Python$ ls
Ch01-Quick_Start                Ch09-File_handling
Ch02-Variable_Fundamentals      Ch10-Class
Ch03-Program_Flow               Ch11-Decorations
Ch04-Lists_Tuples_Dictionaries  Ch12-Data_class
Ch05-Functions_lambdas          Ch13-Database_access
Ch06-Modules                    Ch14-Pytest
Ch07-Interator_Generators
Ch08-Exceptions
(base) student@ab-python3:~/Introduction.to.Python$
```

4. Execute the following two commands:
   a. `cd Ch01-Quick_Start`
   b. `ipython`

```
(base) student@ab-python3:~/Introduction.to.Python$ cd Ch01-Quick_Start/
(base) student@ab-python3:~/Introduction.to.Python/Ch01-Quick_Start$ ipython
Python 3.8.8 (default, Apr 13 2021, 19:58:26)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.22.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

   c. The `In [1]:` is the prompt.
   d. All entries are stored in the `In`(put) list in the sequence of being entered.
   e. Previous input commands can be re-executed by entering the command `exec(In[<number>])` where `<number>` is the number associated with the commands.
      i. Magic commands are not accessible this way.

As a Simple Calculator

1. Execute: `3 + 5`
    a. You should see:

```
In [1]: 3 + 5
Out[1]: 8
```

    `Out[1]` is the list of outputs.

2. Execute: `10 / 3`
    a. You should see:

```
In [2]: 10 / 3
Out[2]: 3.3333333333333335
```

    b. All arithmetic is done with double floats.

3. Execute: `cos( 2 * pi)`
    a. You should see:

```
In [3]: cos(2 * pi)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-902a3b2b0844> in <module>()
----> 1 cos(2 * pi)

NameError: name 'cos' is not defined
```

    b. The "batteries are included" but they are not yet accessible. (Ask the instructor what this means.)
    c. The standard C language math functions are in a separate file, called a module.
    d. To use functions, you have to give the interpreter access to the function. This is called importing and is done with the import statement.

4. Execute: `import math`

5. To find out a little about the module, execute `math?`
    a. You should see:

```
In [5]: math?
Type:        module
String form: <module 'math' from '/Users/arthur/anaconda3/lib/python3.6/lib-dynlo
ad/math.cpython-36m-darwin.so'>
File:        ~/anaconda3/lib/python3.6/lib-dynload/math.cpython-36m-darwin.so
Docstring:
This module is always available.  It provides access to the
mathematical functions defined by the C standard.
```

    b. The `File:` entry tells where the module is kept and is not available in all versions of Python.
    c. The `Docstring` provides some documentation on the module.
        i. The command `pdoc <object>` returns just the docstring associated with the object.
    d. The command <object>?? may give you more information.

6. Execute: `help(math)`
    a. This gives a "man page"-like document.
    b. See something similar to:

```
Help on module math:

NAME
    math

MODULE REFERENCE
    https://docs.python.org/3.6/library/math

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
```

    c. Notice the MODULE REFERENCE has the URL for the official Python documentation.
    d. Exit with a `<Ctrl-D>`

7. Execute: `help(math.cos)`
    a. The imported module (math in this case) creates a new namespace. Thus, to access the cos function, you use `math.cos;` to access the value of pi you use `math.pi`.
    b. See:

```
Help on built-in function cos in module math:

cos(...)
    cos(x)

    Return the cosine of x (measured in radians).
(END)
```

8. Execute: `math.cos(2 * math.pi)`
    a. This returns the cosine of twice the value of pi.
    b. See:

```
In [6]: math.cos(2 * math.pi)
Out[6]: 1.0
```

Introduction to the `dir` Command

1. Execute: `dir(math)`
    a. With a module name as an argument, the `dir` command returns a list of the module's attributes, some of which are executable functions.

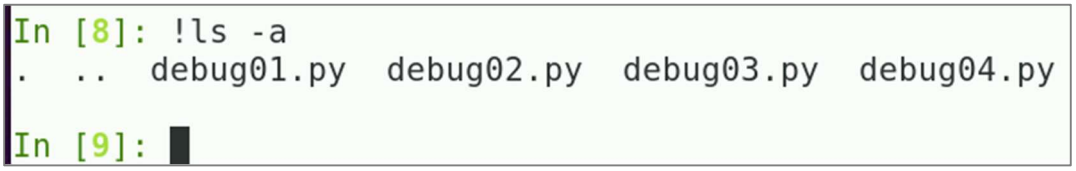
Executing Shell Commands from Inside `ipython`

To execute a shell command from within `ipython`, precede the shell command with an exclamation point.

1. Execute: `!ls`
    a. See:

```
In [7]: !ls
debug01.py  debug02.py  debug03.py  debug04.py
```

2. Execute: `!ls -a`
    a. See:



Magic Commands

Magic commands are commands executed by `ipython`. Their format is `%<command>`. We will see several of these commands while working through this chapter.

1. Execute: `%cd ..`
    a. The change directory command is a magic command because it changes the state of `ipython`. (`cd` is an internal command of the shell.)
    b. To see the current working directory, execute: `%pwd`

2. To see a list of all magic commands, execute: %`lsmagic`

3. To see documentation on a magic command, execute: `%<command>?`

Log Out of `ipython`

1. Enter `exit()` or `<Ctrl-D>` to exit from `ipython`.

2. Log out of `ipython`.

This page is intentionally blank.

Exercise 1.2: Simple I/O

1. Log in and open `ipython`.

2. Ask for a person's name to be input from `stdin`
    a. Assign the object input to the variable `name`.
    b. Execute the statement and answer the question.

3. Write out the statement `Well, hello <name>`.
    a. Use the variable, `name`, from Step 2.
    b. Execute the statement.

4. **DO NOT EXIT FROM `IPYTHON`**. Needed for next exercise.

This page is intentionally blank.

Exercise 1.3: Editing with ipython

Execute: `%history -n`

1. See:

```
In [3]: %history -n
   1: name = input("What is your name? ")
   2: print("Well, hello", name)
   3: %history -n
```

2. `%history` is a magic command (one giving directions to `ipython`) that shows the commands you have entered into `ipython` since this session started.

3. The `-n` adds the line numbers.

Edit the Lines Used in the Previous Exercise (Asking for Name and Saying Hello)

1. Execute: `edit -x 1 2`
   a. See something similar to:

```
In [3]: %history -n
   1: name = input("What is your name? ")
   2: print("Well, hello", name)
   3: %history -n
```

   b. The `-x` is not to execute the program when saved.
   c. The `1 2` is a list of the lines wanted in the edit file.
      i. This is just a list of the lines wanted.
      ii. Use `1:7` to put lines 1 through 6 into the editor. Notice it is `<start><:stop>`.
   d. The default editor is `vi` (`vim`).
      i. This can be changed by changing the environmental variable `EDITOR` to point to editor wanted.

Edit the File to Make It Look Like the Following

```
1 #! /usr/bin/env python
2
3 """
4    filename: first.py
5 """
6
7 name = input("What is your name? ")
8 print("Well, hello", name)
~
```

1. Line 1
    a. This is called the interpreter line.
    b. `#!`, the magic number, saying this file is interpreted by the program that follows.
    c. `/usr/bin/env` tells the shell to look for the interpreter using the directory in the `PATH` environment variable.
    d. `python` is translated to the current version through system setup.
    e. The script still has to be marked as executable with the `chmod` command.

2. Lines 3 through 5
    a. This is called the docstring. It must be the first entry in a Python script file after the interpreter line.
    b. A docstring starts with three double or single quotes and ends with three matching quotes. Spaces and newlines have meaning inside the docstring.
    c. This is used in the Python `help` and `ipython ?/??` documentation systems.

3. Lines 2 and 6
    a. Blank lines have no meaning.

4. Spacing at the beginning of the line is very important; it specifies the structure of the program.
    a. The docstring must be the first thing on the line. *Note:* material inside of the docstring is at the programmers' whim.
    b. The executable lines must be at the beginning of the line. Much, much more in the chapter on functions and the chapter on classes.

Save the Program

1. Execute: <`esc`>`:w first.py`
    a. This saves the file as `first.py`.

2. Execute: `<esc>:q!`
    a. This exits the editor without saving the temporary file.

Checking the Program

1. Execute: `%ls -l first.py`
    a. You should see a long listing of `first.py`.

```
In [25]: !ls -l first.py
-rw-rw-r-- 1 student student 117 Nov 15 23:58 first.py
```

2. Execute: `%cat first.py`
    a. You should see something similar to:

```
In [24]: !cat first.py
#! /usr/bin/env python

"""
   file:  first.py
"""

name = input("What is your name? ")
print("Well, hello", name)
```

Execute the Program Inside of `ipython`

1. Execute: `%run first.py`
    a. You should see:

```
In [26]: %run first.py
What is your name? haha
Well, hello haha
```

Running a Python Script Outside of `ipython` on Linux System

1. Exit from `ipython` (`<Ctrl-D>` or `exit()`).

2. Make sure you are in `Ch01-Quick_Start`.

3. Execute: `python first.py`
    a. This should run the program.

4. Execute: `chmod 755 first.py`
    a. This should allow anyone to execute the program.

5. Execute: `./first.py`
    a. This should execute the program.

Exercise 1.4: Chapter Exercises

Write a Program

1. Write a shell script that asks for a person's favorite color; then waits 5 seconds; then prints out the favorite color and says goodbye.
    a. There is a function in the `time` module called `sleep` which may help in writing this program.

    Hint:

    ```
    import time
    < part of script>
    time.sleep(5)
    <rest of script>
    ```

2. When finished, inform your instructor so they can look at your work.

Debugging Exercises

1. Make sure you are in this directory: `Ch01-Quick_Start`.

2. The program `debug01.py` does not work; figure out what is wrong.

3. The program `debug02.py` does not work; figure out what is wrong. This is tricky.

4. The program `debug03.py` does not work; figure out what is wrong. This is very tricky.

This page is intentionally blank.

Exercise 2.1: String Methods

Strings have a lot of methods—way too many to learn all at one sitting. The objective here is to show you the basics of string manipulation and where the documentation is on strings. If you do not understand the command, do a `help(<command>)`. Use `help("<keyword>")` for keywords or symbols such as "`==`".

1. Log in, open a terminal window, change to the directory `Ch02-Variable_Fundamentals`, and start `ipython`.

2. Execute:
    a. `a = 'Nice ' "Dragon"`
    b. `b = "Nice " + 'Dragon'`
        i. `+` sign is concatenation of `str` objects.
    c. `c = """Nice Dragon"""`
    d. `d = '''Nice  Dragon'''`
        i. Triple quotes accept `<enter for newline>`
        ii. Single and double quotes lose their meaning inside triple quotes.
    e. `a == b`
        i. Should show true.
    f. `a == c`
        i. Should show true.
    g. `a is b`
        i. Same as `id(a) == id(b)`
        ii. Should show false.
    h. `a is c`
        i. Should show false.
    i. Understanding
        i. Each assignment statement makes a different object.
        ii. Single and double quotes must be matched, and you can use double or single quotes interchangeably.
        iii. Triple quotes (docstring) accept `<enter>` for a '`\n`' character and single and double quotes have no meaning.

3. Execute: `dir(a)`

   If `a` is a string, this returns all of the identifiers defined in the class (namespace) `str`.

   a. The `__<string>__` are for symbols or string definitions.
      i. `a.__gt__(b)` is the function for `a > b`.

4. Execute: `help(a.lower)`

5. How would you do a case-insensitive comparison, given the methods of `str`, on the strings `"NICE dragons FINISH Last"` and `'Nice Dragons Finish Last'`?

   Show work:

   _____

6. Given the string `"Nice Dragons Finish Last"`:
   a. Do split the string into a list of words.

7. To check to see if a string can be converted to a base 10 `int`, which method should be used?

   _____

8. If time, read through all of the methods given at https://docs.python.org/3/library/stdtypes.html#textseq. Scroll down to *String Methods*.

Exercise 2.2: Chapter Exercise

Debugging Exercises

1. Make sure you are in this directory: `Ch02-Variable_fundamentals`.

2. The program `debug1.py` does not work; figure out what is wrong.

Change debug01.py

1. Count the number of lines and print it at the end of the output.

What Is Happening Here?

1. Log in, change to directory `Ch02-Variable_Fundamentals`, and start `ipython`.

2. Execute: `a = 0.1 + 0.1 + 0.1`

3. Execute: `a`
   a. What happened here? Why?

4. Execute: `print('%0.10f' % a)`

5. Execute: `b = 0.3`

6. Execute: `b`

7. Execute: `print('%0.20f' % b)`

8. Execute: `a == b`
   a. This should return `False`.

9. Here is the moral:

   The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2**53 per operation.

   This is a problem with all languages and operating systems.

   a. See https://docs.python.org/3/tutorial/floatingpoint.html for a simple explanation.

10. Python included the modules `fractions` and `decimal` for those who need very precise numbers.

Exercise 4.1: List Introduction

A list is a sequence so much of this exercise is just to see how a list is manipulated as a sequence.

1. Log in, open a terminal window, change to `Ch04_Lists_Tuples_Dictionaries`, and start `ipython`.

2. Enter: `q = [ 1, "Plum Lucky", 3.14, 2, "One for the Money", 3 ]`
   a. The `[]` enclose a list.
   b. A list, technically, is a positional order, mutable collection of 0 or more references to other objects.

Slice Operator

1. Enter: `q[1]`
   a. This should return "Plum Lucky".
   b. The index starts at 0.

2. Enter: `q[0:2]`
   a. This should return a list `[1, "Plum Lucky"]`.
   b. All sequence operators work with lists.

3. Enter: `q[1:]`
   a. Returns `[ "Plum Lucky", 3.14, 2, "One for the Money", 3 ]`.

4. Enter: `q[-1]`
   a. Returns `3`.

5. Enter: `q[:3]`
   a. Returns `[1, "Plum Lucky", 3.14]`.

Concatenation, Repeating, Deleting, and Slice Assignment

1. Enter: `a = [ 1, 2, 3 ]`
2. Enter: `b = [ 4, 5, 6 ]`
3. Enter: `a + b`
   a. Returns: `[ 1, 2, 3, 4, 5, 6 ]`
   b. This is concatenation.
4. Enter: `a * 2`
   a. Returns: `[ 1, 2, 3, 1, 2, 3 ]`
   b. This is a form of concatenation called repeating.
5. Enter: `c = a + b`
6. Enter: `c`
   a. Returns: `[ 1, 2, 3, 4, 5, 6 ]`
7. Enter: `del c[5]`
8. Enter: `c`
   a. Returns: `[ 1, 2, 3, 4, 5 ]`.
   b. A list is a mutable object. The `del` operator deletes the reference from the list `c`. Recall that a list is really a sequence of references and what has been done is to remove the reference associated with the index.
9. Enter: `del c[1:3]`
10. Enter: `c`
    a. Returns: `[1, 4, 5]`.
    b. Deletes the objects at indexes 1 and 2.
    c. Indexes are dynamically assigned to the index. The index value for 4 is now 1.
11. Enter: `d = [ 1, 2, 3, 4, 5, 6 ]`

12. Enter: `d[1] = 'a'`

13. Enter: `d`
    a. Returns: `[ 1, 'a', 3, 4, 5, 6 ]`.
    b. `d[1]` is a reference to the object whose value is 2 on line 11. On line 12, the reference is changed to refer to an object with a value of '`a`'.

14. Enter: `d[2:4] = [ 'b', 'c' ]`

15. Enter: `d`
    a. Returns: `[ 1, 'a', 'b', 'c', 5, 6 ]`.
    b. This is done in two steps. First, the `[2:4]` references are removed. Second, starting at index 2, references are inserted into the list.

16. Enter: `d[2:4] = [ "A", "B", "C", "D" ]`

17. Enter: `d`
    a. Returns: `[ 1, 'a', 'A', 'B', 'C', 'D', 5, 6 ]`.
    b. First, the references to '`b`' and '`c`' were deleted, and then the references from the list will be inserted starting at index 2.

This page is intentionally blank.

Exercise 4.2: List of Lists, a Class Exercise

1. Enter: `e = [ 1, 2, 3, 4, 5, 6 ]; e`
   a. What was displayed?

      _____

   b. The semicolon (`;`) allows multiple statements on the same line.

2. Enter: `e[1] = [ 'a', 'b' ]; e`
   a. What was displayed?

      _____

   b. This is not a slice assignment (no colon in the `e[1]`); it is a simple assignment. The reference `e[1]` is changed to be a reference to the object `[ 'a', 'b' ]`.

3. Enter: `e[1]`
   a. What was displayed?

      _____

   b. Remember `e[1]` is a reference. In this case, it references the list object with value `[ 'a', 'b' ]`.

4. Enter: `e[1][1]`
   a. What was displayed?

      _____

   b. The square brackets `([])` are evaluated from left to right. `e[1]` returns `[ 'a', 'b' ]` and the second bracket `([1])` operates on this object and returns '`b`'.
   c. This construct is created by creating a list of lists.
      i. As many dimensions as needed, just a deeper nesting of lists of lists.
      ii. Each dimension has as many elements as wanted as a list has no restrictions on the number of elements.
   d. There are arrays in Python (see the `array` and `numpy` modules; i.e., `array.array` and `numpy.array`).

5. Enter:
```
f = [ [ 1, 2, 3 ],
      [ 4, 5, 6 ],
      [ 7, 8, 9 ] ]
```

6. Enter: `f[1][2]`
    a. What was returned?

       _____

Shared References and Copying

1. Enter: `g = [ 1, 2, 3, 4, 5, 6 ]`

2. Enter: `h = g`
    a. What was returned?

       _____

3. Enter: `h is g`
    a. What was returned?

       _____

    b. `True` should have been returned as both `g` and `h` contain references to the same list.

4. Enter: `g[1] = "Plum Lucky"`

5. Enter: `h[1]`
    a. What was returned?

       _____

    b. Both `g` and `h` refer to the same mutable object. So, changing one changes the other.
    c. It is possible to make a copy of a list.

6. Enter: `j = range(1,7)`

7. Enter: `j`
    a. What is displayed?

       _____

8. Enter: `k = list(j)`
    a. The list function returns a new list.
        i. It does a shallow copy. This is it, only copy the first level of items. This is not a problem with one-dimensional lists.
    b. The sequence operator `j[:]` also returns a new list.

9. Enter: `k is j`
    a. What is displayed?

        _____

10. Enter: `k[1] = "HAHA".`

11. Enter: `j[1]`
    a. What is displayed?

        _____

12. Enter:
```
j2d = [ [ 1, 2, 3 ],
        [ 4, 5, 6 ],
        [ 7, 8, 9 ] ]
```

13. Enter: `k2dr = j2d`

14. Enter: `k2dr is j2d`
    a. What is displayed?

        _____

15. Enter: `k2dc = list(j2d)`

16. Enter: `k2dc  is  j2d`
    a. What is displayed?

        _____

    b. `False` is what should be displayed. And it is true that the references at the top level of `k2dc` are not the same as `j2d`.

17. Enter: `k2dc[1][1] = “HAHA!”`

18. Enter: `j2d[1][1]`
    a. What is displayed?

    _____

    b. `"HAHA!"` should be displayed as list does a shallow copy which means the references in the second dimension of `k2dc` are the same as `j2d`.

19. Enter: `k2dc[1][1] is j2d[1][1]`
    a. What is displayed?

    _____

    b. `True` should be displayed.

20. Enter: `j2d[1][1] = 5`

21. Enter: `import copy`.
    a. This is done to have access to `copy.deepcopy()` which makes a copy of all the dimensions.

22. Enter: `k2dc2 = copy.deepcopy(j2d)`

23. Enter: `k2dc2 is j2d`
    a. What is displayed?

    _____

    b. `False` should be displayed.

24. Enter: `k2dc2[1][1] is j2d[1][1]`
    a. What is displayed?

    _____

    b. `False` should be displayed.

25. Enter: `k2dc2[1][1] = "HAHA!"`

26. Enter: `j2d[1][1]`
    a. What is displayed?

    _____

27. `5` should be displayed.

Optional Exercise 4.3: List Methods

1. Enter: `j = [ 'a', 'b' ]`

2. Enter: `j.append('c'); j`
   a. What is displayed?

      _____

   b. Would this be the same as `j[len(j):] = ['c']`?

      _____

3. Enter: `j.extend(['b', 'a']); j`
   a. What is displayed?

      _____

   b. Would this be the same as `j[len(j):] = ['b', 'a']`?

      _____

4. Enter: `j.insert(3, 'd'); j`
   a. What is displayed?

      _____

   b. Notice that the insert was done before the index, moving the variables starting at the index to the next larger index.

5. Enter: `j.remove('a'); j`
   a. What is displayed?

      _____

   b. Notice that is the first '`a`' that was removed.

6. Enter: `j.index('d')`
   a. What is displayed?

      _____

7. Enter: `j.pop(j.index('d'))`.
    a. What is displayed?

       _____

    b. Remember that without a parameter, `.pop` removes and returns the last item.

8. Enter: `j.count('b')`
    a. What was displayed?

       _____

9. Enter: `j.sort()`.
    a. What was returned?

       _____

    b. Remember that sort does an in-place sort.
    c. The return is '`None`', a built-in constant, meaning nothing was returned.

10. Enter: `j`
    a. What is returned?

       _____

11. Enter: `j.reverse()`
    a. What is returned?

       _____

    b. Again, an in-place reversal.

12. Enter: `j`
    a. What is returned?

       _____

Exercise 4.4: for Compound Statement

This is a reading and executing exercise. It comprises four simple scripts which are designed to explore different aspects of the `for` statement.

1. If necessary, open a terminal window, start `ipython`, and change directories to `Ch04_Lists_Tuples_Dictionaries`.

2. Do a `cat` on `ch04_1_for_simple.py`.

```
#! /usr/bin/env python

"""
        Program:  ch04_1_for_simple.py
        Function:  Explore the for command
"""

p = "Plum Lucky!"

for i in p:
    print(i.upper())

print("That's all folks!")

print("i = ", i)
```

   a. Execute the program.
   b. `p` can be any iterable data type, mutable or non-mutable.
   c. `i` is assigned to `i` – not aliased as in some languages (Perl).
   d. `i` is a new variable and the last value assigned in the `for` loop is preserved when the `for` loop terminates.

3. Do a `cat` on `ch04_2_for_continue.py`.

```
#! /usr/bin/python

"""
Program:  ch04_2_continue.py
Function:  Explore the for command
"""

p = "Plum Lucky!"

for i in p:
    if i in 'aieou': continue
    print(i.upper())

print("That's all folks!")
print("i = ", i)
```

   a. Execute the script and see how `continue` works.

4. `cat` the file `ch04_5_break.py`

```
#! /usr/bin/python

"""
Program:  ch04_5_break.py
Function:  Explore the for command
"""

p = "Plum Lucky!"

for i in p:
    if i in 'aieou': break
    print(i.upper())

print("That's all folks!")
print("i = ", i)
```

   a. Execute the program.
   b. Notice `i` still has the last value assigned to it after the break.

5. `cat` the program `ch04_4_for_assignment.py`.

```
#! /usr/bin/env python

"""
      Program:  ch04_4_for_assignment.py
      Function:  Explore the for command
"""
j = range(5)
print("j = ", j)

for i in j:
    i = i + 10;
    print(i);

print("That's all folks!")

print("i = ", i)
print("j = ", j)
```

   a. The complete syntax for range is:
      `range([start_number],stop_number,[increment_number])`.
      i. The range command returns an iterator that returns the value at `start_number` or `0`, and each time accessed returns the next number stopping at the `stop_number`, using an increment of `1` unless `increment_number` is specified.

This page is intentionally blank.

Exercise 4.5: Assignment and References in Tuples

This is a simple exercise for understanding assignment and references in tuples. Do the exercise in `ipython`.

1. Enter: `t1 = ( 4, "Four to Score", 15.95 )`

2. Enter: `t2 = t1`

3. Enter: `t2 is t1`
   a. This should return `True`.
   b. `t1` is a list of constants and `t2` now points to the same list.

4. Enter: `L1 = [ 1, "One for the Money", 14.95 ]`
   a. Notice `L1` is a list not a tuple.

5. Enter: `L2 = [ 2, "Two for the Dough", 15.95 ]`
   a. Notice `L2` is a list not a tuple.

6. Enter: `t3 = ( L1, L2 ); t3`
   a. What is the value of `t3`? _____

7. Enter: `t4 = t3`

8. Enter: `t4 is t3`
   a. This should return `True`

9. Enter: `t3[1][2] = 20.95`
   a. This should *not* have produced an error as `t3[1][2]` is a list item and is mutable.

10. What is the value of `L2[2]`?
    _____
    a. `L2` and `t3[1]` are actually references to the same object.
    b. Enter: `L2 is t3[1]; L2 is t4[1]`
       i. This should return `True`.
    c. What should `L2 is t4[1]` return? _____

11. Enter: `import copy`

12. Enter: `t5 = copy.deepcopy(t3)`

13. Enter: `t3`; `t5`
    a. Just to see what they are.

14. Enter: `t5 == t3`
    a. This should return `True` as the values are the same.

15. Enter: `t5 is t3`
    a. This should return False as `copy.deepcopy` makes a complete copy of `L2` the object.

Exercise 4.6: zip and Tuples

1. If necessary, open a terminal window, start `ipython`, and `cd` to `Ch04_Lists_Tuples_Dictionaries`.

2. Enter: `L1 = [ 1, "One for the Money", 14.95 ]`

3. Enter: `L2 = [ 2, "Two for the Dough", 15.95 ]`

4. Enter: `t_zip = zip( L1, L2 )`

5. Enter: `t_zip`
   a. What does this return?_____

6. Execute: `t_zip.__next__()`

7. Execute: `t_zip.__next__()`

8. Execute: `t_zip.__next__()`

9. Execute: `t_zip.__next__()`

10. Notice that it is `__next__` and not `next()` which means it is not designed to use directly.

11. `cat` the file `ch04_11_for_zip.py`.

```
#! /usr/bin/env python

"""
Program:  ch04_11_for_zip.py
Function:  To show how zip can be used inside a for
"""

L2 = [ 1, "One for the Money", 14.95 ]
L2 = [ 2, "Two for the Dough", 15.95 ]

for (a, b) in zip( L1, L2):
    print("%20s" % str(a), "\t", "%20s" % str(b))

print("\n\nThat's all folks!")
```

12. Execute the program.

13. Note the tuple assignment in the `for` line.

14. Note that `str(a)` returns a string object and does not return an error if a string is passed to it.

Exercise 4.7: Dictionary Basics

1. Enter:
```
d1 = { "rabbit":"The Tale of Peter Rabbit",
       "squirrel":"The Tale of Squirrel Nutkin",
       "bunny":"The Tale of Benjamin Bunny" }
```

2. Enter: `d1`
   a. Since Python 3.6 has an implementation detail, the order of adding the key/value pairs is kept. An implementation detail means that it may change in the future and should not be relied upon.

3. Enter: `d1["squirrel"]`
   a. If the key does not exist, a `KeyError` exception is raised.

4. Enter: `d1["kitten"] = "The Tale of Tom Kitten"`
   a. If the key "`kitten`" already exists, it is written over.

5. Enter: `d1`

6. Enter: `del d1["rabbit"]`
   a. Removes the entry with the key of "`rabbit`".
   b. If the key does not exist, a `KeyError` is raised.

7. Enter: `d1`

8. Enter: `k1 = ( "fisher", "moppet", "duck" )`

9. Enter:
```
v1 = ("The Tale of Mr. Jeremy Fisher",
      "The Story of Miss Moppet",
      "The Tale of Jemima Puddle-Duck")
```

10. Enter: `d2 = dict(zip( k1, v1 ))`
    a. The function `dict` can take an iterable sequence of two element tuples and return a dictionary. The tuples are in (`key`, `value`) order.

11. Enter: `d2`

12. Enter:
```
d3 = dict( rabbit="The Tale of Peter Rabbit",
           mouse="The Tale of Johnny Town-Mouse" )
```
    a. Notice that rabbit and mouse do not have quotes around them.
    b. This form is called keyword and the key is given before the equal sign as an alphabetic string.

13. Enter: `d3`

14. Enter: `len(d3)`
    a. Should return `2`, the number of keys in the dictionary.

15. Enter: `"rabbit" in d3`
    a. Should return `True`.

16. Enter: `"bunny" not in d3`
    a. Should return `True`.

Exercise 4.8: Dictionary Methods

This is a continuation of Exercise 4.7.

1. `Cat ch04_12_for_dictionary.py`

```
#! /usr/bin/env python

"""
Program:  ch04_12_for_dictionary.py
Function:  Shows how a dictionary interacts with a for
"""

d1 = { "rabbit":"Tale of Peter Rabbit",
       "squirrel":"Tale of Squirrel Nutkin",
       "kitten":"Tale of Tom Kitten",
       "mouse":"Tale of Johnny Town-mouse",
       "bunny":"Tale of Benjamin Bunny" }

for key in sorted(d1):
    print "%12s" % key, "%-25s" % d1[key]

print "\n\nThat's all folks!"
```

2. Execute the script.

3. Change `for key in sorted(d1):` to `for key in d1:` and re-execute.

This page is intentionally blank.

Exercise 5.1: Class Exercise on Scoping

1. If necessary, open a terminal window and `cd` to `~/Introduction.to.Python/Ch05-Functions_lambda`.

2. Much of the rest of this exercise is based upon the script shown below. The numbers on the left are for reference only. Execute: `cat -n ch05_03_function_scope.py` to see in the terminal.

```
 1  #! /usr/bin/python
 2
 3  """
 4  Program:  ch05_03_function_scope.py
 5  Function:  Program for working through scope rules
 6
 7  """
 8
 9  def outer_function():
10        oct = 10
11        print("oct in outer_function 1 =", oct)
12
13        def inner_function():
14              oct = "ABC"
15              print("oct in inner_function =", oct)
16
17        inner_function()
18        print("oct in outer_function 2 =", oct)
19
20 # start of main code
21 oct = 0
22 print("oct in module before =", oct)
23
24 outer_function()
25
26 print("oct in module after =", oct)
27
28 print("That's all folks!")
```

3. Lines 13 through 15 define the function `inner_function`.
   a. Line 14 defines the identifier `oct`. The scope of `oct` is the function body.

4. Lines 9 through 18 define the function `outer_function`.
    a. Line 10 defines the identifier `oct`. The scope of `oct` is the function.
    b. The function `inner_function` is defined inside of `outer_function`.
        i. The `oct` identifier defined in `outer_function` is defined for all statements in the function `outer_function` including the statements of the function `inner_function`.
        ii. The `oct` identifier of `outer_function` is not visible in the function `inner_function` because the identifier `oct` is also defined in `inner_function`. Shadows or covers it.
        iii. If `oct` was not defined in `inner_function`, the `oct` of `outer_function` would be available as an immutable object.

5. Execute the script.
    a. The output should be as shown below:

```
oct in module = 0
oct in outer_function 1 = 10
oct in inner_function = ABC
oct in outer_function 2 = 10
oct in module after = 0
That's all folks!
```

    b. The scope of `inner_function` is enclosed by the scope of `outer_function`. The scope of `outer_function` is enclosed in the module. A module is a file that contains executable Python code. This is referred to as the global scope. This is enclosed in the Python program code.
    c. As each inner scope defines `oct`, the program uses the `oct` defined in the most enclosing scope.

6. Comment out line 14, `oct = "ABC"` and rerun the program again.
    a. The output should be as shown below:

```
oct in module = 0
oct in outer_function 1 = 10
oct in inner_function = 10
oct in outer_function 2 = 10
oct in module after = 0
That's all folks!
```

    b. Since `oct` was not defined in `inner_function`, `oct` in `outer_function` became visible and is used.

7. Comment out line 10, `oct = 10`, and run the script again.
    a. The output should be as shown below:
    ```
    oct in module = 0
    oct in outer_function 1 = 0
    oct in inner_function = 0
    oct in outer_function 2 = 0
    oct in module after = 0
    That's all folks!
    ```
    b. Since `oct` is not defined in `inner_function` or `outer_function`, both of these functions use the `oct` defined in the outermost function, which is the file, called the module in Python.

8. Comment out line 18, `oct = 0`, and run the script again. Yes, `oct` is not defined in the script.
    a. The output should look as follows:
    ```
    oct in module = <built-in function oct>
    oct in outer_function 1 = <built-in function oct>
    oct in inner_function = <built-in function oct>
    oct in outer_function 2 = <built-in function oct>
    oct in module after = <built-in function oct>
    That's all folks!
    ```
    b. After the scope of the script, there is one more place Python looks for the definition of an identifier, the list of built-ins.
        i. In this case, `oct` is a built-in function taking a number and returning it as an octal string.
        ii. In the Python Shell, enter: `import __builtin__`.
        iii. Enter: `dir(__builtin__)`.
            A. This will return a list of all of the built-in predefined identifiers.

Scope Rules Summary (LEGBE)

For an identifier in a function:

```
Local scope searched.
Enclosing functions searched.
Global (module) containing the functions searched.
Built-in identifier searched.
Exception raised.
```

This page is intentionally blank.

Exercise 5.2: Exploring the return Statement

1. If necessary, open a terminal window, start `ipython`, and change directory to `Ch05_functions`.

2. The program below will be used in this exercise. You can see this in the terminal by executing `cat -n ch05_06_function_no_return.py`.

```
 1  #! /usr/bin/env python
 2
 3  """
 4  Program:  ch05_06_function_no_return.py
 5  Function:  Exploring what happens when the function does
 6  not end with a return
 7  """
 8
 9  def simple_function():
10      print("in simple_function")
11      c = 1 + 4
12  #    return c
13  #    return
14
15  return_value = simple_function()
16
17  type_return_value = type(return_value)
18
19  print(" The type of the return value: ", type_return_value
20  print("The value of the return value: ", return_value)
```

3. Run the program.
   a. What is the return type with no return statement?

      _____

   b. What is the value of `return_value`?

      _____

4. Modify the program by uncommenting out line 12.
    a. What is the return type with no return statement?

       _____

    b. What is the value of `return_value`?

       _____

5. Modify the program by commenting out line 12 and uncommenting line 13.
    a. What is the return type with no expression in the return statement?

       _____

    b. What is the value of `return_value`?

       _____

Exercise 5.3: Functions and Shared Objects

1. If necessary, open a terminal window, start `ipython`, and `cd` to `~/Python/Ch05_functions`.

2. This section is based upon the following script.
   Execute: `cat -n ch05_08_functions_shared_objects.py`

```
 1  #! /usr/bin/env python
 2
 3  """
 4   Program:   ch05_08_functions_shared_objects.py
 5  Function:   This is a contrived function but does show how
 6                  mutable and immutable object passing work
 7  """
 8
 9  def add_10( add_10_immutable, add_10_mutable ):
10      print( add_10_mutable is mutable)
11      print( add_10_mutable == mutable)
12      add_10_immutable += 10
13      print("Inside add_10")
14      print("          immutable object   =", add_10_immutable)
15
16      for i in range(len(add_10_mutable)):
17              add_10_mutable[i] += 10
18
19      print("      Local mutable object   =", add_10_mutable)
20      return
21
22  immutable = 10
23  mutable = [1, 2, 3]
24
25  print("Outside add_10")
26  print("      immutable object value =", immutable)
27  print("        mutable object value =", mutable)
28
29  add_10( immutable, mutable )
30
31  print("Outside add_10")
32  print("      immutable object value =", immutable)
33  print("        mutable object value =", mutable)
```

3. Read through the script.
    a. Notice that `mutable` and `add_10_mutable` share the same object. When the objects in `add_10_ mutable` are changed in the function, it is not the identifier of the list that changes but of the object referenced by the list.

4. Execute the program to verify that it is all true.

5. Change lines 16 and 17 to `add_10_mutable = [ x + 10 for x in add_10_mutable]`

6. Execute the new program.

7. Explain the differences between the old and new program.

Optional Exercise 5.4: Class Exercise on Parameter Passing

1. If necessary, open a terminal window, start `ipython`, and `cd` to `Ch05_Functions_lambdas`.

Positional Parameters

```
 1  #! /usr/bin/env python
 1
 2  """
 3  Program:  ch05_11_function_positional.py
 4  Function:  Exploring positional passing
 5  """
 6
 7  def simple_function(a, b, c):
 8      print "I am from simple_function"
 9      print "The value of a: ", a
10      print "The value of b: ", b
11      print "The value of c: ", c
12      return
13
14  print "\nCalled as simple_function( 10, 'my string', 344.1)"
15  simple_function( 10, 'my string',344.1 )
16
17  print "\nThat's all folks!"
```

1. Execute the program.

Default Parameters

1. Read through the script.
    a. Line 15 passes all parameters overriding the defaults.
    b. Line 18 passes only two parameters accepting the default for `c`.

2. *Note:* Once started to accept default parameters, all of the parameters following must also be defaulted.

Tuple Gather

```
 1  #! /usr/bin/env python
 2
 3  """
 4  Program:  ch05_13_function_tuple_gather.py
 5  Function:  Exploring tuple gathering
 6  """
 7
 8  def simple_function(a, b = "string", *c):
 9         print("I am from simple_function")
10         print("The value of a: ", a)
11         print("The value of b: ", b)
12         print("The value of c: ", c)
13         return
14
15  print("\nCalled as simple_function( 10, 'my string', 1, 'a
    string', 3, 4 )")
16  simple_function( 10, 'my string', 1, 'a string', 3, 4 )
17
18  print("\nCalled as simple_function( 10, 1, 'a string', 3, 4 )")
19  simple_function( 10, 1, 'a string', 3, 4 )
20
21  print("\nCalled as simple_function( 10 )")
22  simple_function( 10 )
23
24  #print("\nCalled as simple_function(10, b='new value', 1, 'a
    string', 3, 4)")
25  #simple_function(10, b='new value', 1, 'a string', 3, 4)
26
27  print("\nThat's all folks!")
```

1. Execute the script.

2. Read through the script.
   a. On line 16, the script is called with a bunch of arguments. 10 is assigned to `a`; `'my string'` is assigned to `b`; the rest are placed in a tuple and assigned to `c`.
   b. Line 18 is the same call except the `'my string'` assigned to `b` is left out. What this means is that `b` is assigned the value `1` and the tuple assigned to `c` is one item less.

c. On line 22, the script is called with only one parameter which is assigned to `a`, `b` takes its default, and `c` has no items assigned to it.
d. Line 24 is commented out as it raises an exception. The `b='new value'` cannot be followed by positional variables which must precede it even if they are to be collected in a tuple.

Dictionary Gather

```
 1  #! /usr/bin/env python
 2
 3  """
 4  Program:  ch05_14_function_dictionary.py
 5  Function:  Exploring dictionary gather
 6  """
 7
 8  def simple_function(a, b = "string", **c):
 9         print("I am from simple_function")
10         print("The value of a: ", a)
11         print("The value of b: ", b)
12         print("The value of c: ", c)
13         return
14
15  print("\nCalled as simple_function( 10, 'my string', d1=5,
    d2=3, d3=8 )")
16  simple_function( 10, 'my string', d1=5, d2=3, d3=8 )
17
18  print("\nCalled as simple_function( 10 )")
19  simple_function( 10 )
20
21  print("\nCalled as simple_function(10, b='new value', d1=5,
    d2=3, d3=8)")
22  simple_function(10, b='new value', d1=5, d2=3,d3=8)
23
24  print("\nThat's all folks!")
```

1. Execute the script.

2. On line 16, two positional parameters for `a` and `b` are followed by the keyword parameters which are gathered into a dictionary.

3. On line 19, just the required positional parameter is given. `b` takes the default and `c` has no items assigned to it.

4. On line 22, `b` is given a keyword value followed by many keyword values. Since `b` is a parameter of the function, the `b` keyword is assigned to `b`, and `c` gathers the rest of the keyword parameters into a dictionary.

5. Run `ch05_14_function_dictionary.py`

Pass by Keyword

```
 1  #! /usr/bin/env python
 2
 3  """
 4  Program:  ch05_15_function_pass_by_keyword.py
 5  Function:  Exploring simple pass by keyword
 6  """
 7
 8  def simple_function(a, b, c):
 9        print("I am from simple_function")
10        print("The value of a: ", a)
11        print("The value of b: ", b)
12        print("The value of c: ", c)
13        return
14
15  print("\nCalled as simple_function( c=10, b='my string', a=5
    )")
16  simple_function( c=10, b='my string', a=5 )
17
18  print("\nThat's all folks!")
```

1. Execute the script.

2. Read the script.
    a. On line 15, notice that the order of the keyword values does not have to be in the order of the parameter in the function definition.

Pass Sequence

```
 1  #! /usr/bin/env python
 2
 3  """
 4  Program:  ch05_17_pass_sequence.py
 5  Function:  Exploring sequence expansion
 6  """
 7
 8  def simple_function(a, b):
 9        print("I am from simple_function")
10        print("The value of a: ", a)
11        print("The value of b: ", b)
12        return
13
14  s1 = ('a', 'b')
15  print("\nCalled as simple_function( *s1)")
16  simple_function( *s1)
17
18  s1 = ['a', 'b']
19  print("\nCalled as simple_function( *s1)")
20  simple_function( *s1)
21
22  print("\nThat's all folks!")
```

1. Execute the script.

2. Read through the script.
    a. On line 14, there is a tuple and on line 18 there is a list. The function works with either one. Using `*s` expands the tuple or list.

Function Pass Directory

```
 1  #! /usr/bin/env python
 2
 3  """
 4  Program:  ch05_18_function_pass_dictionary.py
 5  Function:  Exploring key value pairs
 6  """
 7
 8  def simple_function(a, b):
 9      print("I am from simple_function")
10      print("The value of a: ", a)
11      print("The value of b: ", b)
12      return
13
14  d1 = {'a':1, 'b':3}
15  print("\nCalled as simple_function( **d1)")
16  simple_function( **d1)
17
18  print("\nThat's all folks!")
```

1. Execute the script.

2. Read the script.
    a. Note the `**d` breaks the dictionary apart and passes each dictionary element as two elements in the parameter list.

Exercise 5.5: Function as Object

1. Read the script `ch05_20_function_passing.py`
   a. Lines 8 through 25 define a very simple bubble sort. What is interesting is the second parameter which is the name of a function.
   b. The requirements for the function passed to `simple_sort` are that given any two elements of the list to be sorted in the order `a, b` the function returns True if `a` > `b` and otherwise False. The function `string_length_compare(a, b)` does just that using the lengths of strings as the comparator.

2. Execute the program. Notice how the strings are sorted.

3. Modify `string_length_compare` so that the < is now a >.

4. Rerun the program. This should have changed the order of the sort.

```
 1  #! /usr/bin/env python
 2
 3  """
 4  Program:  ch05_20_function_passing.py
 5  Function:  Exploring the names of functions
 6  """
 7
 8  def simple_sort(list_sort, cmp_function):
 9        new_list_sort = list_sort[:]
10
11        def swap( list_in, a, b ):
12              temp = list_in[a]
13              list_in[a] = list_in[b]
14              list_in[b] = temp
15              return
16
17        again = True
18        while again:
19            again = False
20            for i in range(0,len(new_list_sort) - 1):
21                value =
    cmp_function(new_list_sort[i],new_list_sort[i + 1])
22                if value:
23                    swap(new_list_sort, i, i+1)
24                    again = True
25         return new_list_sort
```

Continued on the next page.

```
26  def string_length_compare(a, b):
27          return len(a) > len(b)
28
29  list1 = [ 'abcde', 'xy', 'm', 'rqc', 'jwif' ]
30
31  print("  Variable list1 to be sorted: ", list1)
32  sorted_list1 = simple_sort( list1, string_length_compare )
33  print("Variable list1 after the sort:", list1)
34  print("     Sorted list sorted_list1:", sorted_list1)
35
36  print("\nThat's all folks!")
```

Optional Exercise 5.6: Chapter Exercise

Part I: Programming

1. Program `Ex5_1.py` is only partially completed. Complete as many of the functions as you feel necessary to understand creating functions.

2. Change the function in `ch05_20_function_passing.py` to sort the strings from right to left. That is, if the string is "`abc`", then the major sort is '`c`', the first minor sort is '`b`', and the last minor sort is '`a`'.

Part II: Debugging

There are three files starting with debug in `Ch05_functions`. Read and fix each script.

This page is intentionally blank.

Exercise 6.1: Simple Exceptions

1. If necessary, open a terminal window, then `cd` to `Ch06_Exceptions`.

2. Execute: `cat -n ch06_01-Simple.py`
   a. See:

```
 1  #! /usr/bin/env python
 2  """
 3  File: ch06_01-Simple.py
 4  Function: first pass try-except-else
 5  """
 6
 7  while True:
 8      try:
 9          n = input("Please enter an integer: ")
10          n = int(n)
11      except ValueError as error:
12          print(error)
13          print(error.args)
14          print(error.args[0])
15      else:
16          print("An integer, " + str(n) +
17          ", has been entered.")
18          break
```

   b. Execute: `python ch01_01_01-simple.py`
   c. The `ValueError` is returned when the wrong type is passed to the code.
   d. The `ValueError` in line 11 is a rather complex object. Take it as some magic has been performed and `error` is a tuple extracted from the object containing information about the error. The first element is always an error description string.

3. Execute: `cat -n ch06_02-Simple.py`
    a. See:

```
 1 """
 2    File: ch06_02-Simple.py
 3    Function: show a double except
 4    Note: no Object returned if a tuple of
 5    exceptions
 6 """
 7 import sys
 8
 9 while True:
10     try:
11         n = input("Please enter an integer: ")
12         n = int(n)
13     except ValueError as error:
14         print(error.args[0], file=sys.stderr)
15     except (EOFError, KeyboardInterrupt):
16         print("\nYou must enter an integer to exit!",
file=sys.stderr)
17     else:
18         print("An integer, " + str(n) + ", has been
entered.")
19         break
```

    b. Execute: `python ch06_02-Simple.py`
    c. The `EOFError` is given when an EOF is received.
    d. The `KeyboardInterrupt` is returned when a `<Ctrl-C>` is entered.
    e. When the `except` clause is followed by a tuple of error types, there is no tuple returned with the error data. We will see in the next script how to retrieve the error data.

4. Execute: `cat -n ch06_03-Simple.py`
    a. See:

```
 1 """
 2    File: ch06_03-Simple.py
 3    Function: Introduction to sys.exc_info
 4    sys.exec_info needed for execpt without
 5    error data.
 6 """
 7 import sys
 8
 9 while True:
10     try:
11         n = input("Please enter an integer: ")
12         n = int(n)
13     except ValueError as error:
14         exception_class, exception_string, traceback =
sys.exc_info()
15         print("   error.args[0]: ", error.args[0])
16         print(" exception_class: ", exception_class)
17         print("execption_string: ", exception_string)
18         print("       traceback: ", traceback)
19         print("You must enter an integer to exit!\n",
file=sys.stderr)
20     except:
21         exception_class, exception_string, traceback =
sys.exc_info()
22         print("\n exception_class: ", exception_class)
23         print("execption_string: ", exception_string)
24         print("       traceback: ", traceback)
25         print("You must enter an integer to exit!\n",
file=sys.stderr)
26     else:
27         print("An integer, " + str(n) + ", has been
entered.")
28         break
```

    b. `sys.exc_info()` returns information about the current process when an exception happens. As you can see in line 14 or line 22, it returns the exception class, the exception string, and an object that can be used to create a traceback of the error.

This page is intentionally blank.

Exercise 6.2: User Exceptions

An exploration of simple user exceptions.

1. Enter: `cat ch06_11_user_exceptions.py`

```
 1 #! /usr/bin/env python
 2
 3 """
 4 Program: ch06_11-user_exceptions.py
 5 Function: An exploration of user exceptions
 6 """
 7
 8 import sys
 9 import traceback
10
11 class MyErrors(Exception): pass
12
13 def get_number():
14     number = int(input("Enter a number (10 - 99): "))
15     if number < 10 or number > 99:
16         raise MyErrors("Number must be between 10 and 99")
17     return number
18
19 while True:
20     try:
21         number = get_number()
22         result = 100 / number
23     except  MyErrors as  error_string:
24         print(error_string.args[0])
25     except  (KeyboardInterrupt, EOFError):
26         print("\nQuiting by user request", end="",
file=sys.stderr)
27         break
28     except:
29         # should catch valueError separately
30         exc_type, exc_value, exc_traceback =
sys.exc_info()
31         traceback.print_exception(exc_type, exc_value,
exc_traceback,
32                                   limit=None,
file=sys.stdout)
33     else:
34         print("The value is ", result)
35
36 print("\nGood bye!")
37 sys.exit(0)
```

a. The Exception class does not need to be imported—it is always available.
b. Line 11 creates a simple sub-class of Exception. It does nothing.
c. Line 16 uses the `raise` statement to raise the exception. It is also passing the exception a string to be used as the argument to the exception.
d. Line 23 is the `except:` clause which processes the raised exception. Notice the argument used catch the string passed on Line 11.

2. Answer the following questions without running the program. Confirm your answers by running the program.
   a. What happens when the following is entered?
      i. `<Ctrl-D>` _____
      ii. `<Ctrl-C>` _____
      iii. `25` _____
      iv. `199` _____
      v. `Abc` _____

Exercise 7.1: Importing a Module

1. Log in, open a terminal window, change to the directory `Ch07_Modules`, and start `ipython`. You must start `ipython` from the directory `Ch07_Modules`.

2. `cat` the file `first.py`.

```
 1  # first.py
 2
 3  mod_int = 25
 4  mod_list = [ 1, 2, 3 ]
 5
 6  def ident():
 7          print("From the module ", __name__)
 8          return
 9
10  def add_2( a, b ):
11           return a + b
```

   a. Notice there is no `#!` (shebang) to mark this as an executable file for the OS.

3. In the terminal window, make sure you are in `Ch07_Modules`.

4. Enter: `ls *.pyc`.
   a. The return should tell you there are no such files.
   b. If there are, erase them by entering `rm *.pyc`.

5. To create a method of accessing the attributes of a module, the modules must be imported.
   a. One method of doing this is with the `import` *`module_name`* command.

6. Start: `ipython`

7. Enter: `import first`
   a. Notice that the `.py` suffix was not part of the *`module_name`*.
   b. The module name must have a suffix of `.py`; but it will cause a syntax error if it is part of the `module_name` used in the import command.

8. Enter: `!ls`
    a. The screen should show: `__pycache__ first.py`
    b. The `__pycache__` directory was created by the execution of the import command.
        i. This directory contains `cpython` code (faster loading and executing) for the Python scripts in the modules in the directory.

9. Enter: `!tree __pycache__`
    a. This shows the files and subdirectories under `__pycache__`
    b. The `import` command is an executable command. The first time the command is executed, `import` creates a Python byte-code file for the modules in the directory. This is the file with the suffix `.pyc` (the `c` stands for compiled). This file cannot be read in a standard editor. It is still OS independent and can be distributed just as any other Python file. The file is in a form that is very fast for the Python interpreter to load. It is no faster than the `.py` file after it is loaded into memory. What is saved is the time to convert the `.py` file to the byte-code format and perhaps some time in loading the file if the file is very large.
    c. If this file is available and the time-date stamp is newer than the time-date stamp on the `.py` file, the source code for the module, Python will use this file for the next first import.
        i. A Python interpreter will only import a module once. If there is a second import statement for a module already imported, the module is not imported again <u>even if the module has changed</u>.
        ii. The reload function can be used to force a module to be imported again. The reload function is shown later in this chapter.

10. At this point, the attributes of the module are available as *module_name.attribute*.

11. Python searches an ordered list of directories looking for the module.
    a. Enter: `import sys`
    b. The `path` attribute of the `sys` module contains the list of directories to be searched. The Python interpreter can access the attributes of the module, but the module must be imported to be seen.
    c. Enter: `sys.path`
    d. The return is a list of the directories to search.
    e. The empty string `' '` represents the current directory.

12. Enter: `first.add_2(3,5)`
    a. You should see a return of `8`.

13. Enter: `first.mod_list`
    a. You should see a list `[1, 2, 3]`.

This page is intentionally blank.

Exercise 7.2: Module Documentation

Fixing Help in the module first.

1. Make sure you are in the directory `Ch07_Modules`.

2. Start `ipython` and `cat -n` the file `first_doc.py`.

```
 1  """
 2  Module: first_doc
 3
 4  The first module is for learning how to import attributes
 5  and the problems of importing attributes that are variables
 6  in the module.
 7
 8  """
 9
10  mod_int = 25
11  mod_list = [ 1, 2, 3 ]
12
13  def ident():
14      """
15      Returns a string saying which module being accessed
16      """
17      print("From the module", __name__)
18      return
19
20  def add_2( a, b ):
21      """
22       Returns the + operation on 2 objects of the same
23       type.
24       """
25      return a + b
```

3. Lines 14 through 16, a `docstring` (`"""..."""`) provides the documentation for `ident`. The `docstring` must appear immediately after the `def` line. Recall that spaces and newlines are preserved inside a `docstring`.

4. Lines 20 through 22 provide the documentation for `add_2`.

5. Lines 1 through 6 provide the documentation for the module `first`. This `docstring` must be the first thing in the module file except for lines starting with `#`, which starts a comment.

6. The file `first.py` can be edited. That is a lot of work. The following will do this a little easier.
    a. Enter: `import first`
    b. Enter: `ls first*`
        i. The return should be `first_doc.py`, `first.py`.
    c. Enter: `dir(first)`
    d. Enter: `help(first)`
    e. Enter: `cat first_doc.py`
        i. This should show `first.py` with all the `docstrings` added.
    f. Enter: `cp first_doc.py first.py`
    g. Importing `first` again will do no good as Python does nothing with the second import.
        i. `ipython` could be stopped and started and first imported again. This is time consuming.
    h. Enter:
    ```
    import importlib
    importlib.reload(first)
    ```
        i. This causes Python to reload the module `first` to be reloaded from the source file `first.py`.
        ii. The screen should display:
        ```
        <module 'first' from '…first.py'>
        ```

7. Enter: `dir(first)`

8. Enter: `help(first)`

9. Enter: `help(first.add_2)`
    a. The following should be displayed:

    add_2(a, b)
        Returns the + operation on 2 objects of the same type.

Exercise 7.3: Using a Module

1. Make sure you are in the directory `Ch07_Modules`.

2. `cat -n` the file `ch07_01_module_import.py`.

```
 1  #! /usr/bin/env python
 2
 3  """
 4  Program:  ch07_01_module_import.py
 5  Function:  An introduction to module use
 6  """
 7
 8  import first
 9
10  def simple_function():
11         print("The value of mod_int:", first.mod_int)
12
13  print("Using add_2 from first to add 5 + 3 = ", first.add_2(5,
    3))
14
15  print("Accessing mod_int =", first.mod_int)
16  print("Accessing mod_list =", first.mod_list)
17
18  first.ident()
19
20  simple_function()
21
22  Print("That's all folks!")
```

3. Read through the script.
   a. Notice that access to any identifier in the module `first` is done using `first.`*`identifier_name.`*

4. Run the script.

5. Execute: `cat -n ch07./_02_module_from.py`

```
 1  #! /usr/bin/env python
 2  """
 3  Program:  ch07_02_module_from.py
 4  Function:  An introduction to module use
 5  """
 6
 7  from first import add_2, mod_int, mod_list
 8
 9  def simple_function():
10         print("The value of mod_int:", mod_int)
11
12  print("Using add_2 from first to add 5 + 3 = ", add_2(5, 3))
13
14  print("Accessing mod_int =", mod_int)
15  print("Accessing mod_list =", mod_list)
16
17  #ident()
18
19  simple_function()
20
21  print("That's all folks!")
```

   a. On line 7 is a from *`module`* `import` *`attribute_list`* command. This command adds the *`attribute_list`* to the global namespace of the module (file) executing the command. The identifiers added are initialized with references to the same named attributes in the imported module. Only these attributes are available from the module.
   b. It is as if the following code was executed:

```
import first
add_2 = first.add_2
mod_int = first.mod_int
mod_list = first.mod_list
del first
```

      i. The `del` command removes the module `first` from the global symbol table.

c. Notice the `#ident()` on line 17.
    i. `first.ident` is not defined and if uncommented and the script runs, an exception (error) will be raised. Only the *attribute_list* attributes from the module are available.
d. A small word of caution: Python does no checking when adding the attributes to the namespace; it silently writes over identifiers already in the namespace.

6. It is also possible to use `from first import *` to import all of the attributes from the module.
    a. Placing an underscore(_) as the first character of an attribute tells Python not to import this attribute when * is used for the *attribute_list*.
        i. Using an explicit import will allow access to the attribute.
    b. You can specify which identifiers will be imported with a * with the predefined identifier `__all__`
        i. For example: `__all__ = [ 'add_2', 'mod_int' ]`.
        ii. Only the identifiers in the list will be imported.

This page is intentionally blank.

Exercise 7.4: Using a Package

This is a short introduction to modules in packages.

A Python module is a file ending in `.py` which contains definitions of functions and perhaps other identifiers.

A Python package is a directory containing the file `__init__.py` to identify it as a package which contains Python modules and perhaps Python sub-packages.

1. Change directory to `Ch07_Modules`.
2. Execute: `ipython`
3. Execute: `import numpy as np`
4. Execute: `dir(np)`
5. Look for `random`. It is not a module.
6. Execute: `np.random?`
7. Execute: `help(np.random.random_integers)`
8. If you read the help file, you will see there is a lot of magic going in.
9. Execute: `np.random.random_integers(1,100)`
10. This should return an `np.array` of three numbers between 1 and 100.
11. Execute: `from np.random import random_integers as NPRI`
12. This should create an error saying `np` not found!
13. Execute: `from numpy.random import random_integers as NPRI`
14. Execute: `NPRI(1,100,3)`
15. Execute: `from numpy.random import randint as NPRI`
16. Execute: `NPRI(1,100,3)`

17. *Note:* the new import statement did write over the old statement.

18. What are the differences between `random_integers` and `randint`?

19. Execute: `exit()`

Class Exercise 8.1: Python Classes Introduction

This is an introduction to Python classes. When finished with this chapter, you should be able to use a class or class hierarchy and build moderately complex classes. This should cover about 95 % of what you will encounter in Python code.

This chapter does not cover abstract base classes at all. It does not cover multiple inheritance or slots.

Class Introduction

1. Log in, open a terminal window, and change to directory `Ch08-Class`.

2. Execute: `cat -n Robot_I.py`
   a. You should see:

```
1   class Robot:
2       """
3          This is the first class
4       """
5       pass
```

   b. Line 1
      i. Defines the start of a class definition. Indent all that part of the `class`.
      ii. `Robot` is the name of the class. Typically, a class name starts with a capital letter and is a noun.
   c. Lines 2 through 4
      i. This is the docstring documentation for the class. It must be the first non-blank line entry.
   d. Line 5
      i. `pass` is a keyword meaning nothing additional defined.

Accessing a Class Introduction

1. Open a second terminal window, change to the directory `Ch08-Class`, and start `ipython`.

2. Execute: `from Robot_I import Robot`

3. Execute: `type(Robot)`
   a. You should see:

```
In [2]: type(Robot)
Out[2]: type
```

   b. The response `type` means that a class is the method of defining a new type. Type as in `int, str, or dict`.
   c. This creates a new namespace and a scope with the definition of the class.

4. Execute: `Robot.__dict__`
   a. You should see:

```
In [3]: Robot.__dict__
Out[3]:
mappingproxy({'__dict__': <attribute '__dict__' of 'Robot' objects>,
              '__doc__': '\n       This is the first class\n    ',
              '__module__': 'Robot_I',
              '__weakref__': <attribute '__weakref__' of 'Robot' objects>})
```

   b. `__dict__` is a special attribute of a class which contains all of the identifiers defined in the class namespace.
      i. `__name__` is another special attribute of the class that contains the name of the class.
   c. `__doc__` is the docstring.
   d. `__module__` is the name of the file the class was defined in.
   e. `__weakref__`
      i. A `weakref` is an advanced method of creating a reference to an object—not covered in this course.

Step 1: Creating an Instance

1. Execute: `r1 = Robot()`
   a. This creates an instance, `r1`, of the class Robot.
   b. This is an "empty" – has no attributes – instance.

2. Execute: `type(r1)`
   a. You should see:

```
In [7]: type(r1)
Out[7]: Robot_I.Robot
```

   b. `r1` is a reference to an object of the class Robot as defined in the module (file) `Robot_I`.

3. Execute: `r1.__dict__`
   a. See:

```
In [9]: r1.__dict__
Out[9]: {}
```

   b. `r1` has its own namespace, scope, which exists as long as `r1` exists.

An Instance Attribute

1. Execute: `r1.name = 'Robbie'`
   a. You could have used `setattr(r1, 'name', 'Robbie')`
   b. This creates a new identifier in `r1`'s namespace with a reference to the string `Robbie`.

2. Execute: `r1.__dict__`
   a. See:

```
In [12]: r1.__dict__
Out[12]: {'name': 'Robbie'}
```

   b. Confirmation that a new name was created in the local namespace for `r1`. That is a new instance of Robot would not have the name attribute.

3. Execute: `Robot.__dict__`
    a. See:

```
In [13]: Robot.__dict__
Out[13]:
mappingproxy({'__dict__': <attribute '__dict__' of 'Robot' objects>,
              '__doc__': '\n       This is the first class\n    ',
              '__module__': 'Robot_I',
              '__weakref__': <attribute '__weakref__' of 'Robot' objects>})
```

    b. Confirmation that it was not created in `Robot`'s namespace.

4. Execute: `print( r1.name )`
    a. See:

```
In [21]: print(r1.name)
Robbie
```

    b. `r1` says go to the `r1` namespace. The dot (`.`) says go to (access) the object with reference on the right. The print accesses the value of the object and sends the value to standard out.

5. Execute: `getattr( r1, "name", "Not defined")`
    a. See:

```
In [28]: getattr(r1, 'name', 'Not defined')
Out[28]: 'Robbie'
```

    b. `r1` is the namespace.
    c. `name` is the identifier.
    d. `'Not defined'` is the default return if the identifier is not defined.

6. Execute: `del r1.name`
    a. `del` removes an identifier from the namespace.
    b. `r1` is the namespace.
    c. `name` is the identifier.

7. Execute: `r1.__dict__`
    a. See:

```
In [24]: r1.__dict__
Out[24]: {}
```

    b. Confirmation the identifier has been removed from the namespace.

A Class Attribute

1. Execute: `Robot.count = 1`
    a. This could also have been done with `setaddr`.

2. Execute: `Robot.__dict__`
    a. See:

```
In [26]: Robot.__dict__
Out[26]:
mappingproxy({'__dict__': <attribute '__dict__' of 'Robot' objects>,
              '__doc__': '\n       This is the first class\n    ',
              '__module__': 'Robot_I',
              '__weakref__': <attribute '__weakref__' of 'Robot' objects>,
              'count': 1})
```

    b. Notice the new identifier defined.

3. Execute: `print( Robot.count)`
    a. See:

```
In [31]: print(Robot.count)
1
```

4. Execute: `r1.__dict__`
    a. See:

```
In [24]: r1.__dict__
Out[24]: {}
```

    b. Confirmation the identifier is not in the `r1` namespace.

5. Execute: `print(r1.count)`
    a. See:

```
In [32]: print(r1.count)
1
```

    b. `r1` is a sub-namespace (scope) of the class scope `Robot` and the normal scope rules apply.

Step 2: Adding Attributesto Robot

1. Exit from `ipython` in first terminal, but do not close the terminal.

2. In a second terminal, execute: `cat -n Robot_II.py`
    a. See:

```
1  class Robot:
2      """
3         This is the second class
4      """
5      count = 0
6      def __init__(self, name = None):
7          self.name = name
8          type(self).count += 1
```

    b. Line 5
        i. Defines a class variable count. The data here is about the class number of `Robot`s created, not anything about the instance.

    c. Lines 6 through 8
        i. Define a function `__init__`.
        ii. Because it is defined inside of a class definition, it is called a method.
        iii. `__init__` is a special method.
        iv. It is called by the class function, in this case `Robot()`.
        v. Parameters passed to `Robot()` are passed to `__init__`.
        vi. `self` is the reference to the instance being made. It is always the first parameter of `__init__`.

d. Line 7
   i. `self.name` is initialized with the value passed in.
   ii. It is stored in the instance object.

e. Line 8
   i. This increases the class variable by 1 because 1 new `Robot` has been created.
   ii. Notice that the access is `type(self).count`. If just `count` was used without the `type(self)`, an instance variable would have been created in the object, which is not what was wanted.
   iii. `type(self)` evaluates to `Robot` which could have been used. Used `type(self)` as it works better with inheritance.

3. In the first terminal, start `ipython`.

4. Execute: `from Robot_II import Robot`

5. Execute: `print(Robot.count)`
   a. See:

```
In [2]: print(Robot.count)
0
```

6. Execute: `r2 = Robot('Robbie2')`

7. Execute: `print(r2.name)`
   a. See:

```
In [3]: print(r2.name)
Robbie2
```

8. Execute: `print(r2.count)`
   a. See:

```
In [5]: print(r2.count)
1
```

Deleting an Instance

1. An instance can be removed with the `del` built-in command.
    a. Execute: `del r2`

2. Execute: `r2.name`
    a. This should return the following error:

```
In [37]: r2.name
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-37-7e6a26f9fc0b> in <module>()
----> 1 r2.name

NameError: name 'r2' is not defined
```

3. Execute: `Robot.count`
    a. See:

```
In [4]: Robot.count
Out[4]: 1
```

    b. If count is to be the number of instances, need a way to subtract 1 from count when object is deleted.

`__del__` for Deleted Objects

1. You do not have to worry about memory clean-up, automatic garbage collection will take care of it.

2. You do have to worry about class variables or perhaps connections.

3. Exit from `ipython`.

4. Execute: `cat -n Robot_III.py`
    a. See:

```
 1  class Robot:
 2      """
 3         This is the third class
 4      """
 5      count = 0
 6      def __init__(self, name = None):
 7          self.name = name
 8          type(self).count += 1
 9      def __del__(self):
10          type(self).count -= 1
```

    b. Lines 9 and 10
        i. Define the method `__del__` which is called each time an instance is deleted.
        ii. Line 10 corrects the count.

5. Start `ipython`.

6. Execute: `from Robot_III import Robot`

7. Execute: `Robot.count`
    a. See:

```
In [2]: Robot.count
Out[2]: 0
```

8. Execute: `r3= Robot('Robbie3')`

9. Execute: `r3.count`
    a. See:

```
In [4]: r3.count
Out[4]: 1
```

10. Execute: `del r3`

11. Execute: `Robot.count`
    a. See:

```
In [6]: Robot.count
Out[6]: 0
```

12. Exit from `ipython`.

Adding A Method

1. Execute: `cat -n Robot_IV.py`
    a. See:

```
 1  class Robot:
 2      """
 3         This is the fourth class
 4      """
 5      count = 0
 6      def __init__(self, name = None):
 7          self.name = name
 8          type(self).count += 1
 9      def __del__(self):
10          type(self).count -= 1
11      def say_hi(self):
12              print(self.name, + ', says "hi!"')
```

    b. Line 12
        i. Use `self.name`, not `name`, why?

2. Start `ipython`.

3. Execute: `from Robot_IV import Robot`

4. Execute: `r4 = Robot('Robbie4')`

5. Execute: `r4.say_hi()`
    a. See:

```
In [3]: r4.say_hi()
Robbie4, says "hi!"
```

6. Exit from `ipython`.

Individual Project

Create and test a class with the following characteristics.

1. Name of class should be `Animals`.
2. An accurate count of the animals should be available.
3. Each animal should have a name.
4. Each animal should make a noise (generic animal sound for now).

This page is intentionally blank.

Class Exercise 8.2: More on Methods

The Problem

So far, we have just saved the attribute of a robot. What happens if we have to do some checking or conversion before saving the retrieving the value of the attribute?

Getters and Setters

1. Execute: `cat -n Robot_V.py`

```
 1  class Robot:
 2      """
 3         This is the fifth class
 4      """
 5
 6      count = 0
 7
 8      def __init__(self, name = None):
 9          self.set_name(name)
10          type(self).count += 1
11
12      def __del__(self):
13          type(self).count -= 1
14
15      def say_hi(self):
16          print(self.name + ', says "hi!"')
17
18      def set_name(self, name):
19          if name:
20              self.__name = name
21          else:
22              self.__name = "Name not Given"
23
24      def get_name(self):
25          return self.__name
```

   a. Line 9
      i. The attribute is not directly set. `set_name` is called to store the name in the instance object.
      ii. *Note:* Do not pass `self` in. This is done by Python as this is a method call.

b. Line 20
   i. The `__name` is a private variable. This means it is not accessible except by code in the class.

2. Start `ipython`.

3. Execute: `from Robot_V import Robot`

4. Execute: `r5 = Robot()`

5. Execute: `r5.__name`
   a. This should return the following error:

```
In [3]: r5.__name
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-8a3ffd2b043b> in <module>()
----> 1 r5.__name

AttributeError: 'Robot' object has no attribute '__name'
```

6. Execute: `r5.get_name()`
   a. You should see:

```
In [5]: r5.get_name()
Out[5]: 'Name not Given'
```

7. Exit from `ipython`.

Properties

1. Execute: `cat -n Robot_VI.py`

```
1.class Robot:
2.    """
3.       This is the fifth class
4.    """
5.
6.    count = 0
7.
8.    def __init__(self, name = None):
9.        self.__set_name(name)
10.        type(self).count += 1
11.
12.    def __del__(self):
13.        type(self).count -= 1
14.
15.    def say_hi(self):
16.        print(self.name, ', says "hi!"')
17.
18.    def __set_name(self, name):
19.        if name:
20.            self.__name = name
21.        else:
22.            self.__name = "Name not Given"
23.
24.    def __get_name(self):
25.        return self.__name
26.
27.    name = property(__get_name, __set_name)
```

   a. Line 20
      i. This sets `name` up as a property. When `name` is to be read, Python will use `__get_name`. When name is to be written to, `__set_name` will be used.

2. Start `ipython`.
3. Execute: `from Robot_VI import Robot`
4. Execute: `r6 = Robot()`
5. Execute: `r6.name`
    a. See:

```
In [3]: r6.name
Out[3]: 'Name not Given'
```

6. Execute `r6.name = 'Robbie6'`
7. Execute `r6.__get_name()`
    a. This will produce the following error:

```
In [5]: r6.__get_name()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-5-d52325fa24da> in <module>()
----> 1 r6.__get_name()

AttributeError: 'Robot' object has no attribute '__get_name'
```

    b. The double-underscore in front of the function identifier makes the function hidden, just like it did for the variable identifier.

8. Execute `r6.name`
    a. See:

```
In [6]: r6.name
Out[6]: 'Robbie6'
```

9. Exit from `ipython`.

`@staticmethod`

1. Execute: `cat -n Robot_VII.py`

```
 1 class Robot:
 2     """
 3        This is the seventh class
 4     """
 5
 6     __count = 0
 7
 8     def __init__(self, name = None):
 9         self.__set_name(name)
10         type(self).__count += 1
11
12     def __del__(self):
13         type(self).__count -= 1
14
15     def say_hi(self):
16         print(self.name + ', says "hi!"')
17
18     def __set_name(self, name):
19         if name:
20             self.__name = name
21         else:
22             self.__name = "Name not Given"
23
24     def __get_name(self):
25         return self.__name
26
27     @staticmethod
28     def robot_count():
29         return Robot.__count
30
31     name = property(__get_name, __set_name)
```

   a. Lines 27 through 29
      i. Method returns the current count of active robot instances.
   b. Line 27
      i. The `@staticmethod` is a decorator that allows the method to be called using the class identifier even if an object instance doesn't exist.
      ii. A method decorated with `@staticmethod` can only access class attributes.

2. Start `ipython`.

3. Execute: `from Robot_VII import Robot`

4. Execute: `Robot.robot_count()`
    a. See:

```
In [1]: from Robot_VII import Robot

In [2]: Robot.robot_count()
Out[2]: 0
```

5. Exit from `ipython`.

`@classmethod`

1. Execute: `Robot_VIII.py`
    a. See:

```
1  class Robot:
2      """
3         This is the eighth class
4      """
5      __count = 0
6
7      def __init__(self, name = None):
8          self.__set_name(name)
9          type(self).__count += 1
10
11     def __del__(self):
12         type(self).__count -= 1
13
14     def __eq__(self, other):
15         return self.name == other.name
16
17     def say_hi(self):
18         print(self.name + ', says "hi!"')
19
20     def __set_name(self, name):
21         if name:
22             self.__name = name
23         else:
24             self.__name = "Name not Given"
25
26     def __get_name(self):
27         return self.__name
28
29     @staticmethod
30     def robot_count():
31         return Robot.__count
32
33     @classmethod
34     def count_robots(cls):
35         return cls.__count
36     name = property(__get_name, __set_name)
```

2. `@classmethod` is a method that is shared among all objects of that class. The decorator will cause the class name to be passed as the first parameter.

3. Start `ipython`.

4. Execute: `from Robot_VIII import Robot`

5. Execute: `r8 = Robot()`

6. Execute: `r8.robot_counts()`
    a. See:

```
In [3]: r8.count_robots()
Out[3]: 1
```

7. Exit from `ipython`.

Optional Individual Exercise

1. Add a weight attribute to your animal class.

2. Make sure that the weight is greater than 10 and less than 100.

Exercise 8.3: Single Inheritance

Base Class

1. Log in, open a terminal window, and change to `Ch08-Class/Inheritance`.
2. Execute: `cat -n Robot.py`
   a. See:

```
 1 class Robot:
 2     """
 3        This is the base class
 4     """
 5
 6     __count = 0
 7
 8     def __init__(self, name = None):
 9         self.__set_name(name)
10         type(self).__count += 1
11
12     def __del__(self):
13         type(self).__count -= 1
14
15     def __set_name(self, name):
16         if name:
17             self.__name = name
18         else:
19             self.__name = "Name not Given"
20
21     def __get_name(self):
22         return self.__name
23
24     name = property(__get_name, __set_name)
25
26     @staticmethod
27     def robot_count():
28         return Robot.__count
29
30     @classmethod
31     def count_robots(cls):
32         return cls.__count
33
34     def __str__(self):
35         return "Robot with name  " + self.__get_name()
36
37     def say_hi(self):
38         print(self.name + ', says "hi!"')
```

b. There is a problem with this code that we will not see until we use this as a base class and create subclasses.

Subclass `Stutter.py`

1. Execute: `cat -n Stutter.py`
   a. See:

```
 1 from Robot import Robot
 2
 3 class Stutter(Robot):
 4     def __init__(self, name=None, times=1):
 5         super().__init__(name=name)
 6         self.__times = times
 7
 8     def say_hi(self):
 9         print('Hi! ' * self.__times, end="")
10         print("Said " + super().name)
11
12    def say_goodbye(self):
13         print('goodbye! ' * self.__times, end="")
14         print("Said " + super().name)
```

   b. Line 3
      i. The `(Robot)` is how inheritance is specified.
      ii. The `Robot` class must be available in Stutter's namespace. This is done in Line 1.
   c. Line 4
      i. The `super()` calls the first ancestor (the parent class). Notice that you do not pass `self` in this case but do pass all of the parameters needed to properly initialize an object of the parent class (`name` is the only such parameter).
   d. Line 8
      i. This overrides the `say_hi` in the base class.
      ii. The MRO (Method Resolution Order) specifies the order Python searches for methods.
         A. To see this:
            1. Open `ipython`.
            2. Execute: `from Stutter import Stutter`
            3. Execute: `Stutter.__mro__`
   e. Line 12 defines a new method available in Stutter.

Testing Inheritance

1. Start `ipython`.
2. Execute: `from Stutter import Stutter`
3. Execute: `Stutter.__mro__`
4. Execute: `s1 = Stutter(name='S1', times=3)`
5. Execute: `s1.name`
6. Execute: `s1.name = 'haha'`
7. Execute: `s1.say_hi()`
8. Execute: `Stutter.count_robots()`
9. Exit from `ipython`.

A Second Layer of Inheritance

1. Start `ipython`
2. Execute: `from Heavy import Heavy`
3. Execute: `!cat -n Heavy.py`

```
 1  from Stutter import Stutter
 2
 3  class Heavy(Stutter):
 4      def __init__(self, name=None, times=3, weight=0):
 5          super().__init__(name=name, times=times)
 6          self.__set_weight(weight)
 7
 8      def __set_weight(self, weight):
 9          if weight > 300:
10              self.__weight = 300
11          else:
12              self.__weight = weight
13      def __get_weight(self):
14          return self.__weight
15    weight = property(__get_weight, __set_weight)
```

This page is intentionally blank.

Exercise 9.1: Class Exercise on Character File I/O

1. If necessary, open a terminal window, and `cd` to `Ch09-Text_file_handling`.

2. Execute: `less -N ch09_01_file_read.py` which is shown below.

```
 1 #! /usr/bin/env python
 2
 3 """
 4     Program: ch09_01_file_read.py
 5     Function: First of several script to explore
 6               opening and reading from a file.
 7 """
 8
 9 import sys
10
11 work_file = input( "Enter file to read: " )
12 if work_file == "":
13     print("Could not read from", work_file, file=sys.stderr)
14     sys.exit(1)
15
16 file_read = open( work_file, "r")
17
18 for line in file_read:
19     line = line[:-1]
20     print (line)
21
22 file_read.close()
23
24 print("That's all folks!")
25 sys.exit(0)
```

3. Read through the script
   a. Why line 19? _____

4. Execute `./ch09_01_file_read.py` and answer the prompt with the file name `file1.txt`. The full or relative paths work here.
    a. The output should look like:

```
Enter file to read: file1.txt
This is line 1 from file1.txt
This is line 2 from file1.txt
This is line 3 from file1.txt
This is line 4 from file1.txt
This is line 5 from file1.txt
This is line 6 from file1.txt
This is line 7 from file1.txt
This is line 8 from file1.txt
This is line 9 from file1.txt
That's all folks!
```

5. On the command line of the terminal window:
    a. Enter: `./ch09_01_read_file.py`
        i. The program should blow up.

```
$ ./ch09_01_file_read.py
Enter file to read: ugabuga
Traceback (most recent call last):
  File "./ch09_01_file_read.py", line 16, in <module>
    file_read = open( work_file, "r")
IOError: [Errno 2] No such file or directory: 'ugabuga'
$
```

    b. The program only checks for empty input; an `open` statement should always be wrapped in a `try except` block with a user-friendly (or at least helpful) error message.
        i. Wrap the `open` line in a `try except` block and test.

6. Execute: `less -N ch09_02_file_write.py`
    a. See:

```
 1 #! /usr/bin/env python
 2
 3 """
 4     Program: ch09_02_file_write.py
 5     Function: Something on write to a file
 6 """
 7
 8 import sys
 9
10 work_file = input( "Enter file to write to: " )
11 if work_file == "":
12     print("No file entered", file=sys.stderr)
13     sys.exit(1)
14
15 try:
16     file_write = open( work_file, "w")
17 except IOError as err:
18     print("Error: ", err, file=sys.stderr)
19     sys.exit(1)
20 except:
21     print( "Unknown error with file", work_file, file=sys.stderr)
22     sys.exit(1)
23
24 while True:
25     try:
26         line = input("Enter line: ")
27     except EOFError:
28         break
29     except:
30         print("Unknown error with input", file=sys.stderr)
31         file_write.close()
32         sys.exit(1)
33     else:
34         file_write.write(line + "\n")
35 k
36 file_write.close()
37 print("\nThat's all folks!")
38 sys.exit(0)
```

7. Execute: `./ch09_02_file_write.py`
    a. Use `haha` as the file name. Enter some text and press ENTER. Either enter more text or press CTRL+D to exit.

8. Execute: `cat haha`. You should see your text.

9. Questions!
    a. On line 34, why add the new line?
    b. Would a new line have to be added if print was used?
        i. Rework with print to test.