Python 3 Programming

Exercise Guide

Python 3 programming - Contents

1

Introduction to Practicals

The practical exercises are designed to supplement the technical training material in this course. Many of the questions introduce further details of functions discussed during the course.

Each practical describes overall objectives and any special requirements for the questions. Most of the questions will require you to look up information in the on-line manual pages for their completion.

Optional exercises may be included at the end of a section marked "**If time allows**". These questions are usually more complex. You are not normally expected to attempt these questions unless you have completed the first set of questions before the end of the time allotted for the practicals.

Many of the exercises will require you to write complete programs, although these will be fairly short. In some cases, additional material such as pre-written functions will be provided. The instructions for the exercises will indicate when this has been done.

If the course is carried out on QA premises, you have the option to use Microsoft Windows or a Linux VM. The username and password for the Linux VM is `qa`.

Online Directory Structure

On Windows: all of the files mentioned here are stored in a directory called **`labs`**. This is normally on your C: drive, but may reside elsewhere, particularly for courses not carried out on QA premises. Your instructor should clarify the exact location.

On Linux: all the files should be in your home directory. If using the supplied Linux VM, this will be `/home/qa`.

Solutions

Sample solutions to all the questions are provided online, in a subdirectory called **`solutions`**, and in printed form after the questions for each chapter. Please remember that, as in many programming tasks, there are several ways to approach and solve most problems. Just because yours is different to ours does not mean that your answer is wrong!

You may of course just examine and run the solution provided without writing any code. You will not get as much out of the course if you do that, however. Make sure you understand the solution if you do not attempt to answer a question.

Eclipse

Eclipse is installed on Windows and Linux for delegates who are already familiar with this IDE.

The setup for Eclipse with the PyDev plug-in should be the same on Windows and Linux. There are two workspaces, projects and solutions, on Windows these are under `C:\Labs` and on Linux under `/home/qa`. Within these there is a project for each chapter.

Introduction to Python 3

Objectives

To become familiar with the environment used for this course, and some basics of Python 3.

Reference Material

Chapter 1, Introduction to Python, in particular the slide "Anatomy of a Python script". It is also a good idea to start getting familiar with the online Python documentation.

Overview

These are simple exercises designed to get you used to your environment. Solution code may be found after the questions, and online in the **`solutions`** folder.

Questions

1. Which version of python is being used?
2. Using an editor, create a new script called **`ex1.py`**. Within this script create two variables, one containing your first name and another containing your last name. Display them using **`print`** with a space between each one.

   Run the script:

   a) From a command-line

   b) From IDLE
3. We have a simple module for displaying a playing card, called **`Showcard`**. It has a function called **`display_file`** which takes one parameter: the name of a gif file. The playing card filenames are of the following format:

   BMP*n*.GIF

   Where *n* is a number between 1 and 52, indicating the ordinal number of the card in the pack. Note that on Linux this is case sensitive.

   Write a Python program called **`pickacard.py`** which uses this module to display a single card. Prompt the user for a number, as follows:

   ```
   number = input("Pick a card (1-52):")
   ```

   Then construct a filename from number, and then display the card by passing the filename to **`Showcard.display_file`** as a parameter.

   Do not worry about out-of-range numbers for the time bein

If time allows…

4. The card display currently times out after 5 seconds. You may wish to adjust this by calling `Showcard.set_timeout(`*number_of_seconds*`)`

Solutions

From IDLE, open the file using the menus File/Open, which will create a new window displaying the source code. Run that code from the new window using Run/Run module (or by pressing <F5>).

1. Which version of python is being used?

```
$ python -V
Python 3.3.0
```

2. Using an editor, create a new script called **ex1.py**. Within this script create two variables, one containing your first name and another containing your last name. Display them using **print** with a space between each one.

   Run the script:

   a) From a command-line

   b) From IDLE

   Our solution has this code in it:

```
first = 'Fred'
last  = 'Bloggs'
print (first,last)
```

   Notice that the separating the two variables with a comma inserts a space for us.

3. Write a Python program called **pickacard.py** which uses the **Showcard** module to display a single card. Here is our solution:

```
import Showcard

number = input("Pick a card (1-52):")

filename = "BMP"+number+".GIF"
Showcard.display_file(filename)
```

Fundamental Variables

Objectives

To experiment with some of the basic variable types within Python, and some of their operations.

Reference Material

Chapter 2 Fundamental Variables, and the Python Manuals.

Questions

1. This exercise carries out some basic operations on variables
   a) Create a new script called ex2.py
   b) Create two variables, one containing your first name and another containing your last name. Display them using `print`.
   c) Now transfer these variable values into a list and display the list.
   d) Take the variables and now store the values in a dictionary, using keys 'first' and 'last'. Display the dictionary values.

   …and execute the script ex2.py.

2. Now we will try some object methods. Create a Python script (call it ex2.2.py if you like) with the following line:

   ```
   var = input("Please enter a value: ")
   ```

   This is an easy way of outputting a prompt to the console and getting a reply. The variable **`var`** is a reference to that reply, which is a *string*.

   Now print the following:

   a) The value of `var` as upper case.
   b) The number of characters in `var` (this does not require a method).
   c) Does it contain numeric characters? (try the `isdecimal()` method).

If time allows…

3. The height of a projectile (y) from a gun (ignoring air resistance) is given as:

$$y = y_0 + x \tan\theta - \frac{gx^2}{2(v_0 \cos\theta)^2}$$

where:

g : Acceleration due to gravity: 9.81 m/s squared

v_0 : the initial velocity m/s

θ : (theta) elevation angle in radians

x : the horizontal distance travelled

y_0 : height of the barrel (m)

Write a Python program to answer the following question:

At a barrel height of 1m, after a horizontal distance of 0.5m, an elevation of 80 degrees, and an initial velocity of 44 m/s, what is the height of the projectile?

To convert degrees (`deg`) to radians use:

```
theta = deg * pi/180
```

You will need to import some math methods:

```
from math import pi, tan, cos
```

There will be a further *if time allows* question which expands on this code after the Collections chapter.

4. Create a new program called **`F1.py`**, it will explore some of the mathematics involved in managing a Formula 1 racing car.

The task of this program (at first), is to answer a question:

Q. "During a race of **45** laps, what is the minimum fuel requirement?"

You will need to know the fuel consumption found during the race qualifying, which is **2.25** kg for each lap.

5. In this exercise we will make a few more modifications to F1.py. First we will add an extra fuel load, and then we are going to calculate the lap time based on the weight of fuel, which naturally decreases each lap.

a. In the previous exercise we worked out the minimum fuel requirement for a 45 lap race, and stored this in a variable named `FuelRequirement`. To fill the tank with the absolute minimum amount of fuel would be foolhardy, and not allow the drivers any margin for manoeuvre. Typically a car will carry an extra 50% for contingency (multiply the minimum by 1.5), so what fuel will be carried by our fictional F1 car at the start of the race?

Modify your `F1.py` program to calculate this.

b. You might think it odd that fuel is measured in kilograms rather than litres or gallons. This is because the weight of fuel is critical to the way a Formula One car performs.

The qualifying lap time was 80.45 seconds, but that was with only 5kg of fuel: **each 10 kg of fuel increases the lap time by 0.35 seconds**.

What will be the lap time for the first lap with all the required fuel on board?

Solutions

Question 1

```
# Create two variables, one containing your first name
first = 'Fred'
# and another containing your last name.
last  = 'Bloggs'
# Display them using print.
print(first,last)

# Now transfer these variable values into a list
names = [first, last];
# display the list
print(names)

# Transfer these variable values into a dictionary,
# using keys 'first' and 'last'.
mydict = {'first' : first,
          'last'  : last}

# Display the values.
print(mydict['first'], mydict['last'])
```

Question 2

```
var = input("Please enter a value: ")

# The value of var as upper case
print(var.upper())

# The number of characters in var
print(len(var))

# Does it contain numeric characters?
print(var.isdecimal())
```

If time allows…

Question 3

```
from math import pi, tan, cos

# 1 Mile per Hour = 0.44704 Meters per Second

g     = 9.81          # Acceleration due to gravity m/s
                      # squared
v0    = 44            # The initial velocity m/s
theta = 80 * pi/180   # elevation angle in radians
x     = 0.5           # the horizontal distance travelled
y0    = 1             # height of the barrel (m)

y = y0 + x*tan(theta) - (g*x**2)/(2*((v0*cos(theta))**2))

print('Height:',y,'m')
```

Questions 4 & 5

```
# This race requires 45 laps, how much fuel is required?
FuelPerLap = 2.25
Laps = 45
FuelRequirement = Laps * FuelPerLap

# Typically a car will carry an extra 50% for contingency
Fuel = FuelRequirement * 1.5
print("Full fuel load:",Fuel,"kg")

# The qualifying lap time was 80.45 seconds
# However, that was with only 5kg of fuel
# Each 10 kg of fuel decreases the lap time by 0.35
           seconds

QLapTime = 80.45

# Theoretical initial lap time would be
TLapTime = QLapTime - (0.35/10) * 5

print("Theoretical initial lap time:",TLapTime)

LapOneTime = TLapTime + ((Fuel/10) * 0.35)
print("Lap one time:", LapOneTime, "seconds")
```

Flow Control

Objectives

To use the flow control structures of Python, and to gain familiarity in coding based on indentation! That does take a little practice. We will also be using a couple of modules from the Python standard library.

Reference Material

Chapter 3 Flow Control. The Python online documentation for the glob and os modules may also be useful. The final question uses the built-in range(), the syntax is given in the list of built-ins at the end of Chapter 01 Introduction to Python.

Questions

1. Using a **`for`** loop, display the files in your home directory, with their size
    a) Use the skeleton file **Ex3.py**
    b) Get the directory name from the environment using os.environ, HOMEPATH on Windows HOME on Linux (we have done that part for you, notice the test of `system.platform`).
    c) Construct a portable wildcard pattern using `os.path.join` (we have done that part for you as well)
    d) Use the `glob.glob()` function to obtain the list of filenames
    e) Use `os.path.getsize()` to find each file's size
    f) Add a test to only display files that are not zero length
    g) Use `os.path.basename()` to remove the leading directory name(s) from each filename before you print it.

2. Write a Python program that emulates the high-street bank mechanism for checking a PIN. Keep taking input from the keyboard (see below) until it is identical to a password number which is hard-coded by you in the program.

To output a prompt and read from the keyboard:

```
supplied_pin = input("Enter your PIN:")
```

Restrict the number of attempts to 3 (be sure to use a variable for that, we may wish to change it later), and output a suitable message for success and failure. Be sure to include the number of attempts in the message.

3. Write a Python program to display a range of numbers by steps of -2.
    a) Prompt the user at the keyboard for a positive integer using:
    ```
    var = input ("Please enter an integer: ")
    ```

    b) Validate the input (`var`) to make sure that the user entered an integer using the `isdecimal()` method. If the user entered an invalid value, output a suitable error message and exit the program.
    c) Use a loop to count down from this integer in steps of 2, displaying each number on the screen until either 1 or 0 is reached. For example, if the integer 16 (validated) is entered, the output would be:
    ```
    16
    14
    12
    10
    8
    6
    4
    2
    0
    ```
    and if 7 is entered, the output would be:
    ```
    7
    5
    3
    1
    ```

    You will need to look-up the **`range()`** built-in in the online documentation, pay particular attention to the *stop* parameter.

If time allows…

4. If a year is exactly divisible by 4 but not by 100, the year is a leap year. There is an exception to this rule. Years exactly divisible by 400 are leap years. The year 2000 is a good example.

   Write a program that asks the user for a year and reports either a leap year or *not* a leap year. (*Hint*: x % y is zero if x is exactly divisible by y.) Test with the following data:

1984	is a leap year	1981	is NOT a leap year
1904	is a leap year	1900	is NOT a leap year
2000	is a leap year	2010	is NOT a leap year

   Use the following to ask the user for a year:

   ```
   year = int(input ('Please enter a year :'))
   ```

5. Take a look at the code template provided in **weekday.py**. Complete this program, to ask for a date in DD/MM/YYYY format and print out the day of the week for this date.

   There is a formula, called *Zeller's Congruence*, which calculates the day of the week from a given day, month and year. Zeller's congruence is defined as follows:

   ```
   z=(1+d+(m*2)+(3*(m+1)/5)+y+y/4-y/100+y/400) % 7
   ```

   where d, m and y are day, month, year and z is an integer (0 = Sun .. 6 = Sat).

   But with the following adjustments *before* use in the formula:

   If month is 1 or 2 and year is a leap year, subtract 2 from day.
   If month equals 1 or 2 and year is not a leap year, subtract 1 from day.
   If month is 1 or 2, add 12 to month.

   Your program should print out the name of the day (e.g. Monday), e.g.:

1/1/1980	Tuesday	9/8/1982	Monday
25/12/1983	Sunday	31/5/1989	Wednesday
2/2/1990	Friday	29/2/1992	Saturday

.

Solutions

Question 1

Here is our portable solution:

```
import sys, glob, os

# Get the directory name
if sys.platform == 'win32':
    dir = os.environ['HOMEPATH']
else:
    dir = os.environ['HOME']

# Construct a portable wildcard pattern
pattern = os.path.join(dir,'*')

# Use the glob.glob() function to obtain the list of
filenames
for filename in glob.glob(pattern):

   # Use os.path.getsize to find each file's size
   size = os.path.getsize(filename)

   # Only display files that are not zero length
   if size > 0:
      print(os.path.basename(filename), size, 'bytes')
```

Question 2

There are several solution to this question as well. Here is ours:

```
pin     = '0138'
limit   = 3
counter = 0


while counter < limit:
    supplied_pin = input("Enter your PIN:")
    counter += 1
    if supplied_pin == pin:
        break


if supplied_pin != pin:
    print("You had",limit,"tries and failed!")
else:
    print("Well done, you remembered it!")
    print("... and only after",counter,"attempts")
```

Question 3

Validate the input (`var`) to make sure that the user entered an integer using the `isdecimal()` method. If the user entered an invalid value, output a suitable error message and exit the program.

```
var = input("Please enter an integer: ")

if not var.isdecimal():
    print("Invalid integer:",var)
    exit(1)
```

Use a loop to count down from this integer in steps of 2, displaying each number on the screen until either 1 or 0 is reached.

```
for var in range(int(var),-1,-2):
    print(var)
```

If time allows…

Question 4

```
y = int(input ('Please enter a year :'))

if y % 4 == 0 and (y % 400 == 0 or y % 100 != 0):
    print("Leap Year")
else:
    print("NOT a leap year")
```

Question 5

```
date = input("Please enter date (DD/MM/YYYY): ")
d,m,y = date.split('/')
d = int(d)
m = int(m)
y = int(y)

if m == 1 or m == 2:
    m += 12
    if y % 4 == 0 and (y % 400 == 0 or y % 100 != 0):
        d -= 2
    else:
        d -= 1

z = 1 + d + (m * 2) + (3 * (m + 1) // 5) + y + y//4 - \
    y//100 + y//400

z %= 7

days =
["Sun","Mon","Tues","Wednes","Thurs","Fri","Satur"]

print(days[z]+'day')
```

If you are interested in validating dates, checking days, months, or years, then look at the **calendar** module in the Python standard library.

String Handling

Objectives

To consolidate string manipulation in Python. This includes further practice at general Python constructs, such as loops.

Reference Material

Chapter 04 String Handling, and Chapter 03 Flow Control. You will also need the Python online documentation.

Questions

1. Open the script **`sep.py`** in a text editor. You will see a string defined called 'Belgium'. Add code to print:
   a) A line of hyphens the same length as the Belgium string, followed by
   b) The string with the comma separators replaced by colons '`:`'., followed by
   c) The population of Belgium (the second field) **plus** the population of the capital city (the forth field). Hint: the answer should be 11183818.
   d) A line of hyphens the same length as the Belgium string

2. In this exercise much of the code has been written for you! Open the script **greek.py** in **IDLE** and run it there (use <F5>) – **do not use Windows cmd.exe** because the character set used cannot handle the Greek characters.

   The **`try`** and **`except`** blocks are examples of exception handling in case it is run under cmd.exe – we will cover these later in the course.

   The script has a list of names for the characters in the Greek alphabet, and it displays each one within a loop. The character itself is generated using the **`chr()`** built-in function (look it up if you are curious). Within Unicode, Greek lowercase characters start at position 0x03b1 (alpha).

   Currently the output is a bit messy and tame, like this:

```
Alpha : α
Beta : β
Gamma : γ
Delta : δ
Epsilon : ε
Zeta : ζ
Eta : η
```

The task in this question is to replace the existing **`print`** function with another (using **`format`**) which displays for each character:

The hex value of the character (**`pos`**)

The character name (**`cname`**), left justified, maximum 12 characters

A colon separator

The lowercase Greek character (**`char`**)

The uppercase Greek character

Your output should look something like this:

```
0x3b1 Alpha       : α A
0x3b2 Beta        : β B
0x3b3 Gamma       : γ Γ
0x3b4 Delta       : δ Δ
0x3b5 Epsilon     : ε E
0x3b6 Zeta        : ζ Z
0x3b7 Eta         : η H
```

and so on..

If time allows…

3. Take a look at the file **messier.txt** in the labs directory, which contains details of celestial "Messier" objects. It consists of a number of columns for each object, identified by the 'M' number. The columns are as follows:

   M number Common name Type Constellation

   Note that many have no common name.

   Read the file in a for loop in the following manner:

   ```
   for line in open('messier.txt'):
       if not line: break
       # The text is available through variable 'line'
   ```

   Ignore lines that do not start with 'M'. Print the fields from each line delimited with '|' characters. Where there is no common name, use 'no name'. Ignore any lines not beginning with a Messier number. For example:

   ```
   |M1|The Crab Nebula|Supernova remnant|Taurus|
   |M2|no name|Globular cluster|Aquarius|
   |M3|no name|Globular cluster|Canes Venatici|
   ```

   Hint: the header on the file should assist in getting the field positions.

Solutions

Question 1

a) A line of hyphens the same length as the Belgium string, followed by

b) The string with the comma separators replaced by colons ':'., followed by

c) The population of Belgium (the second field) **plus** the population of the capital city (the forth field). Hint: the answer should be 11183818.

If you did this:

```
print (items[1] + items[3])
```

then you would have got string concatenation, and an apparently very large number! You need to change each value to an int.

d) A line of hyphens the same length as the Belgium string

```
items = Belgium.split(',')
print ('-' * len(Belgium))              # a)
print (':'.join(items))                 # b)
print (int(items[1]) + int(items[3]))   # c)
print ('-' * len(Belgium))              # d)
```

Question 2

```
greek =
['Alpha','Beta','Gamma','Delta','Epsilon','Zeta',
 'Eta','Theta',Iota','Kappa','Lambda','Mu',
 'Nu','Xi','Omicron','Pi','Rho','Sigma final',
 'Sigma','Tau','Upsilon','Phi','Chi','Psi','Omega']

#Format required:
#    The hex value of the character
#    The character name (cname), left justified,
#    maximum 12 characters
#    A colon separator
#    The lowercase Greek character
#    The uppercase Greek character

pos = 0x03B1
for cname in greek:
    try:
        char = chr(pos)
        #print(cname,":",char)
        print("{0:#x} {1:<12s}: {2} {3}".
              format(pos,cname,char,char.upper()))
        pos += 1
    except UnicodeEncodeError as err:
        print (cname, 'unknown')
```

If time allows…

Question 3

```
for line in open('messier.txt'):
    if not line: break
    if line.startswith('M'):
        # Slice each field
        Mnum = line[:6].rstrip()
        Name = line[6:40].rstrip()
        if not Name: Name = 'no name'
        Type = line[40:64].rstrip()
        Cons = line[64:].rstrip()

        print("|"+Mnum+"|"+Name+"|"+Type+"|"+Cons+"|")
```

Collections

Objectives

To understand the use and syntax of containers in Python 3. We will also compare different ways to access a list.

Reference Material

Chapter 5 Collections, and Chapter 3 Flow Control.

Questions

1. What is wrong with this?

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg']
cheese += 'Oke'
```

How should `'Oke'` be added to the end of the cheese list?

2. What is going on here? Can you explain the output?

```
tup = 'Hello'
print (len(tup))
```

Prints 5

```
tup = 'Hello',
print(len(tup))
```

Prints 1

3. We need to do some maintenance on a dictionary of machines:

```
machines = {'user100': 'yogi',
            'user1'  : 'booboo',
            'user2'  : 'rupert',
            'user3'  : 'teddy',
            'user4'  : 'care',
            'user5'  : 'winnie',
            'user6'  : 'sooty',
            'user7'  : 'padders',
            'user8'  : 'polar',
            'user9'  : 'grizzly',
            'user10' : 'baloo',
            'user11' : 'bungle',
            'user12' : 'fozzie',
```

```
'user13' : 'huggy',
'user14' : 'barnaby',
'user15' : 'hair',
'user16' : 'greppy'}
```

Don't type this in! It should be available for you to edit in Ex5.3.py in the `labs` directory (your home directory on Linux).
Without altering the initial definition of the dictionary, write code that will implement the following changes:

a) user14 no longer has a machine assigned

b) The name of user15's machine is changed to 'cinnamon'

c) user16 is leaving the company, and a new user, user17, will be assigned his machine

d) user4, user5, and user6 are all leaving at exactly the same time, but their machine names are to be stored in a list called `unallocated`. Hint: pop in a loop.

e) user8 gets another machine called 'kodiak' in addition to the one they already have.

f) Print a list of user, with their machine, in any order. Print each user/machine pair on a separate line.

g) Print a list of unallocated machines, sorted alphabetically.

4. This exercise will compare various ways of accessing a list. The script **Ex5.4.py** creates a list called '**`words`**' from a file of the same name. Each item in the list contains a single word. The script calls user-written functions to find the position of the word 'Zulu' in the list, each function using a different searching technique.

   Each function is called a number of times in a loop, and timings are displayed. If you run the script right now it should display zero times - your task is to write the search technique code.

   Do not worry about the syntax for functions (**`def`**), we will cover that later in the course, but make sure that the **`global`** statements are intact.

   The `return` statement in a Python function allows us to return any object from the function (as it does in many other languages).

   IMPORTANT: Your statements within the functions should, nay, **must**, have consistent indentation (which is four spaces).

Continued on the next page…

a) Open the script **Ex5.4.py** using a text editor. Look for the comments marked `### TODO`, these indicate where you are asked to add code.

   The first function to complete is **`brute_force()`**. In this function use a counting '**`for`**' loop to search sequentially for the word 'Zulu'. When found, break out of the loop and return the word's position, plus 1 (which is the line number in the file).

   Run the script to test it. If you are unsure about the result, uncomment the print statement to display the line number returned: it should be 45400. That applies to most of the other parts to this exercise as well.

b) The next function to complete is rather less code. Recall the **`index()`** method which may be called on a list. It returns the position of the first item found with the given value. Call this in the **`index()`** function, returning the position plus 1. Test the script, is **`index()`** faster than brute force?

c) Just for fun (!) we will now time the `in` keyword. We cannot get the line number from this, but it will be constructive in our comparisons of search methods. Return 1 if 'Zulu' is in `words`, and 0 if it is not.

d) Here we would like you to create a dictionary called **`words_dict`** where the keys are the words, with each value being the position in the `words` list. Look for:

   ```
   ### TODO: Create a dictionary called words_dict
   ```

   A simple `for` loop will suffice for this. Notice that the dictionary creation should be after the `start_timer()` call. Now implement the **`dictionary()`** function to return the value for the key 'Zulu', plus 1.

Which technique wins? You might like to run the script several times to get more meaningful comparisons, or increase the internal loop count (a global called `LOOP_COUNT` near the top of the script).

If time allows...

Try the timing test again but search for a word near the front of the list, for example "aback". What difference does it make?

You should see that the search mechanisms are faster, but using a dictionary should take (or or less) the same time.

If time allows…

5. This exercise is a development of a previous optional exercise to show the trajectory of a projectile, ignoring air resistance. If you did not complete that exercise, take the code from `02 Fundamental variables/traj.py`.

   We are going to plot the trajectory of a projectile onto a graph. You will need two lists, one called `x_axis` and one called `y_axis`.

   Write a loop for values of `x` (horizontal distance travelled) from zero in increments of 0.5. Append each value of `x` to `x_axis`.

   Within the loop, calculate a value for `y` (height of the projectile) for each increment of `x`, and append each value of `y` to `y_axis`.

   Exit the loop when y drops to zero or lower.

   a. We are going to use the `matplotlib` module to plot the graph. You will need to ensure that `numpy` (a pre-requisite) and `matplotlib` are installed.

      Here is the code to plot the graph:

```
import matplotlib.pyplot as pyplot

pyplot.ylabel('Height m')
pyplot.xlabel('Distance m')
pyplot.plot(x_axis, y_axis)
pyplot.show()
```

   b. The graph is not very realistic because the x and y units are not the same base. Take a look at the pyplot method ylim() and figure out a limit for y that will show a realistic trajectory plot.

6. This exercise gives an example of iterating through a dictionary.

   Write a program to calculate random share prices for some fictional companies and display them all continually, with a two second delay between each display.

   Here is a skeleton, as in **`Shareprices.py`**:

```
import time
import random

SharePrices = {'Global Motors':50,
               'Big Blue Inc.':50,
               'Gates Software':50,
               'Banana Computers':50}

# Update stock prices with random price changes

while True:

    # TODO: Iterate through the dictionary,
    # updating each share price (sp) to:
    # max(1.0,
    #   sp * ( 1 + ((random.random() - 0.5)/0.5) * 0.05))
```

```
    # TODO: print each company and its share price

    # Print a blank line between
    print()

    # pause for 2 seconds
    time.sleep( 2 )
```

Printing the company name and share price is an opportunity to practice your `format` statements!

.

Solutions

1. What is wrong with this?

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg']
cheese += 'Oke'
```

If we print the variable `cheese` we see:

```
['Cheddar', 'Stilton', 'Cornish Yarg', 'O', 'k', 'e']
```

The string 'Oke' is a sequence, and using += on a list has broken it down into its constituent parts. It should have been:

```
cheese.append('Oke')
```

2. What is going on here? Can you explain the output?

```
tup = 'Hello'
print (len(tup))
```

Prints 5

That should be no surprise, 5 is the number of characters in 'Hello'.

```
tup = 'Hello',
print (len(tup))
```

Prints 1

This is rather more surprising, but did you notice the trailing comma? That extra comma meant that we created a tuple. The `len()` built-in then reported the number of items in that tuple, which is one.

3. There are several solutions, here is one:

```
machines = {'user100': 'yogi',
            'user1'  : 'booboo',
            'user2'  : 'rupert',
            'user3'  : 'teddy',
            'user4'  : 'care',
            'user5'  : 'winnie',
            'user6'  : 'sooty',
            'user7'  : 'padders',
            'user8'  : 'polar',
            'user9'  : 'grizzly',
            'user10' : 'baloo',
            'user11' : 'bungle',
            'user12' : 'fozzie',
            'user13' : 'huggy',
            'user14' : 'barnaby',
            'user15' : 'hair',
            'user16' : 'greppy'}
```

a) user14 no longer has a machine assigned

```
machines['user14'] = None
```

b) The name of user15's machine is changed to 'cinnamon'

```
machines['user15'] = ' cinnamon'
```

c) user16 is leaving the company, and a new user, user17, will be assigned his machine

```
machines['user17'] = machines['user16']
del machines['user16']
```

d) user4, user5, and user6 are all leaving at exactly the same time, but their machine names are to be stored in a list called unallocated.

```
unallocated = []
for user in ('user4', 'user5', 'user6'):
    unallocated += [machines.pop(user)]
```

e) user8 gets another machine called 'kodiak' in addition to the one they already have.

```
machines['user8'] = [machines['user8'],'kodiak']
```

f) Print a list of all the users, with their machines, in any order.

```
for kv in machines.items():
    print (kv)
```

g) Print a list of unallocated machines, sorted alphabetically.

```
print ("Unallocated machines:",sorted (unallocated))
```

4. These are our results; yours will be different because you are will be running on different hardware.

```
Brute_force : 1.217 seconds
Index       : 0.265 seconds
In          : 0.234 seconds
Dictionary  : 0.031 seconds
```

Several methods offer an improved access time. Remember though that using a dictionary requires a preamble overhead which may only be worth the effort with many searches. The figures are also data-dependant; longer search keys will have an effect on performance.

The code is as follows:

```
...
############################################################
# TODO: USER FUNCTIONS
# Each function should return the line number
def brute_force():
    global words
    for pos in range(0,len(words)):
        if words[pos] == 'Zulu':
            break

    # return the line number
    return pos+1
```

Here is an alternative implementation of `brute_force`, using `enumerate`:

```
def brute_force2():
    global words
    for pos, word in enumerate(words):
        if word == 'Zulu':
            break

    # return the line number
    return pos+1
```

The timings are exactly the same as using `range`, but it is more Pythonic.

```
def index():
    global words
    return words.index('Zulu') + 1

def dictionary():
    global words_dict
    return words_dict['Zulu'] + 1

...

# Create a dictionary from the words list
i = 0
start_timer()

# TODO: Create a dictionary called words_dict
for key in words:
    words_dict[key] = i
    i = i + 1

for i in range(0,LOOP_COUNT):
    line = dictionary()

end_timer("Dictionary")
print ("Dictionary line number:",line)
line = 0
```

If time allows…

Question 5

```
from math import pi, tan, cos

g     = 9.81          # Acceleration due to gravity
                      #    m/s squared
v0    = 44            # The initial velocity m/s
theta = 80 * pi/180   # elevation angle in radians
x     = 0.5           # the horizontal distance travelled
y0    = 1             # height of the barrel (m)

# Initial values for the graphic
y = y0
x = 0.0
x_axis = []
y_axis = []

while y > 0:
    x = x + 0.1
    y = y0 + x*tan(theta) - \
        (g*x**2)/(2*((v0*cos(theta))**2))

    print('x = %.1f m, y     = %.1f m' % (x,y))
    x_axis.append(x)
    y_axis.append(y)

# Graph
import matplotlib.pyplot as pyplot

pyplot.ylabel('Height m')
pyplot.xlabel('Distance m')

# Optional realism
pyplot.ylim(-1,max(max(x_axis),max(y_axis)))

pyplot.plot(x_axis, y_axis)
pyplot.show()
```

Question 6

```
import time
import random

SharePrices = {'Global Motors':50,
               'Big Blue Inc.':50,
               'Gates Software':50,
               'Banana Computers':50}

# Update stock prices with random price changes

while True:

    for key,sp in SharePrices.items():
        SharePrices[key] = max(1.0,
                               sp * ( 1 + ((random.random() –
                               0.5)/0.5) * 0.05))
        print("{:<18s} ${:05.2f}".
              format(key,SharePrices[key]))

    print()
    time.sleep( 2 )
```

Regular Expressions

Objectives

To become familiar with some of Python's regular expression tools, and continuing practice with Python syntax and string handling.

Reference Material

Primarily Chapter 6 Regular Expressions, but you may also need to refer to Chapter 4 String Handling. The online documentation for the **re** module may also be useful.

Questions

1. Write a Python script that will perform *basic* validation and formatting of UK postcodes.

   The postcodes are read from file postcodes.txt. A skeleton script, **Ex6.py**, contains the necessary file handling code - you fill in the gaps at the **TODO (a)** comments. This script is also used for the second exercise (below), so ignore the TODO(b) comments for the time being.

   Blank lines should be ignored.

   The following formatting is to be done:

   Remove all new-lines, tabs and spaces

   Convert to uppercase

   Insert a space before the final digit and 2 letters

   Print out all the reformatted postcodes

   Hints:

   Keep it simple; don't try to do all the formatting on one line of code!

   Read the TODO (a) comments in the code skeleton - ignore those for part (b) for the time being.

   There are several ways to insert the space, the simplest is to use re.sub and a back-reference.

   Put regular expressions into raw strings.

2. Now extend your script so that it performs *basic* pattern validation of each postcode (**TODO (b)** comments)

   The input lines are only to contain a postcode, with no other text.

   The format of a UK postcode is as follows:

   One or two uppercase alphabetic characters

   followed by: between one and two digits

   followed by: an optional single uppercase alphabetic character

   followed by: a single space

   followed by: a single digit

   followed by: two uppercase alphabetic characters

   Alphabetic characters are those in the range A-Z.

   Print out all the reformatted postcodes, indicating which are in error, and a count of valid and invalid codes at the end.

   Hints:

   Do the formatting first, and then test the resulting pattern

   Read the TODO (b) comments in the code skeleton

   Use a raw string for your regular expression

   The test file has 25 valid and 5 invalid postcodes.

If time allows…

3. The area and district part of the postcode (the part before the space) can be validated by looking in file **validpc.txt**. This also records the country the area is in. We have not done the File IO chapter yet, but this is how you read an entire file into a list, one line per element:

   ```
   valid = open('validpc.txt').read().splitlines()
   ```

   Modify your code to search for the area-district captured from your regular expression, and output which country the postcode belongs to. We suggest the following steps:

   a) Read **`validpc.txt`** and store it into a dictionary, where the key is the area-district and the value is the country. So, using the statement given above, you need to write a '`for`' loop to generate the dictionary:

   ```
   for txt in valid:
       # Split up the line around a comma, etc…
   ```

   b) Alter your main regular expression to capture the relevant part of the postcode.

   c) If the postcode matches the RE, look it up in the dictionary and report which country it belongs to, or an error message if it is not there.

Solutions

```
import re

infile = open ('postcodes.txt', 'r')

valid   = 0
invalid = 0

for postcode in infile:
    # Ignore empty lines
    if postcode.isspace(): continue

    # (a): Remove newlines, tabs and spaces
    postcode = re.sub('[ \t\n]', '', postcode)

    # (a): Convert to uppercase
    postcode = postcode.upper()

    # (a): Insert a space before the final digit and 2 letters
    postcode = re.sub('([0-9][A-Z]{2})$', r' \1', postcode)

    # Print the reformatted postcode
    print (postcode)

    # (b) Validate the postcode, returning a match object
    m = re.search (
            r"^[A-Z]{1,2}[0-9]{1,2}[A-Z]? [0-9][A-Z]{2}$",
            postcode)
    if m:
        valid = valid + 1
    else:
        invalid = invalid + 1

infile.close()

# (b) Print the valid and invalid totals
print (valid, "valid codes and", invalid, "invalid codes")
```

If time allows:

Here is the additional code:

```
    # Part c
    valid_dict = {}
    valid = open('validpc.txt').read().splitlines()
    for txt in valid:
        line=txt.split(',')
        valid_dict[line[0]] = line[1]
    valid=None
    # end of valid_dict initialisation

...

    # Note the extra parentheses
    m = re.search (
        r"^([A-Z]{1,2}[0-9]{1,2}[A-Z]?) [0-9][A-Z]{2}$",
        postcode)
    if m:
        valid = valid + 1
        # Part c
        area_district = m.groups()[0]

        if area_district in valid_dict:
            print (postcode,"is in",valid_dict[area_district])
        else:
            print (postcode,"not found")
    else:
        invalid = invalid + 1
```

Data Storage and File Handling

Objectives

To use some of the Python 3 file handling methods, as well as the pickle and gzip modules.

Reference Material

Primarily Chapter 7 Data Storage and File Handling, but also Chapter 3 Flow Control, Chapter 4 String Handling, and Chapter 5 Collections. Background information on pickle and gzip is available in the online documentation.

Questions

1. Write a Python script to list all the unused port numbers in the /etc/services file between 1 and 200

   Steps:

   Become familiar with the input file - view it first

   Write the main code to read the services file one line at a time

   Use string functions or a regular expression to:

   Ignore lines starting with a # comment character

   Ignore lines that just consist of "white-space"

   /etc/services has several columns separated by white-space

   - Use split or a regular expression to isolate the port/protocol field
   - Use another split or regular expression to isolate the port number
   - Don't forget to stop at port number 200!
   - Note that many port numbers have > 1 entry

   On Windows the file is in 'C:\WINDOWS\system32\drivers\etc\services', or in 'C:\WINNT\system32\drivers\etc\services'.

   Many port numbers have more than one entry in the file, but you may assume they are in order.

   Hints:

   open the file
   Read the file line-by-line using a `for` loop
   Consider using a set or a dictionary to hold the port numbers.
   Be careful of comparing strings and int - you will have to convert the port number to an int

2. Using the data in **country.txt**, construct a Python dictionary where the country name is the key and the other record details are stored in a list as the value. Store (pickle) this dictionary into a file (call it country.p).

   Notice the size of the file compare to the original, and then change the program to use gzip.

3. Now write a program which reads the pickled dictionary and displays it onto the console.

   If time allows, convert your pickle to use a shelve.

If time allows…

4. This exercise uses **messier.txt**, which was used in a previous optional exercise (you do not need to have completed that exercise to do this one).

   This file contains details of Messier celestial objects that are identified by a Messier number, the first field in the file.

   The aim is to access the records in the file randomly, using `seek()`. First construct an index (could be in a list or a dictionary) which consists of the file position (use `tell()`) of each record. The key is the first field, the Messier number, which is prefixed M (ignore any lines that do not start with 'M').

   Now prompt the use to enter a Messier number, with or without the 'M', and display the record for that celestial object.

5. You may recall and exercise from Chapter 5 Collections that timed various ways of searching for the word 'Zulu', using the program **Ex5.4.py**. The fastest technique by far was to use a dictionary

   Modify **Ex5.4.py** to use a shelve, preferably by copying the code from your dictionary implementation and modifying the copy. If you did not complete that exercise then use the solution code in `solutions/05 Collections`.

   You will find that the shelve is considerably slower than other techniques. However place an additional `start_timer()`/`end_timer()` around the shelve creation (including loading the data into the dictionary). This should give two sets of times for using a shelve, loading and searching (which is repeated in a loop).

   Where is the biggest overhead? Is this a reasonable test?

Solutions

Question 1

This solution uses regular expressions and sets. A common mistake with this approach is to forget to convert the captured port number to an int, required since `range` returns ints.

```
import re

file = r'C:\WINDOWS\system32\drivers\etc\services'

ports = set()
for line in open(file):
   m = re.search(r'(\d+)/(udp|tcp)',line)
   if m:
       port = int(m.groups()[0])
       if port > 200: break
       ports.add(port)

print(set(range(1,201)) - ports)
```

Questions 2 & 3

```
import pickle
import gzip

country_dict = {}

for line in open('country.txt') :
    row = line.split(',')
    name = row.pop(0)
    country_dict[name] = row

outp = gzip.open('country.p', 'wb')
pickle.dump(country_dict, outp)
outp.close()

# Using a shelve
import shelve
db = shelve.open('country')
for country in country_dict.keys():
    db[country] = country_dict[country]
db.close()

db = shelve.open('country')
print (db['Belgium'])
db.close()
```

If time allows…

Question 4

```
# Construct an index
Index = []
fh = open('messier.txt')

while True:
    line = fh.readline()
    if not line: break
    if line.startswith('M'):
        num = line[1:6].rstrip()
        Index.append(fh.tell() - len(line))

while True:
    num = input("Enter a Messier number(0 to exit): ")
    if num.startswith("M"):
        num = int(num[1:])
    else:
        num = int(num)

    if num < 1: break
    num = num - 1

    fh.seek(Index[num])
    print(fh.readline())
```

Question 5

```
def shelvefunc():
    global shelve_dict
    return shelve_dict['Zulu'] + 1

##########################################

i = 0
start_timer()
import shelve
shelve_dict = shelve.open('shelve_dict')

for key in words:
    shelve_dict[key] = i
    i = i + 1
end_timer("Shelve creation")

start_timer()
for i in range(0,LOOP_COUNT):
    line = shelvefunc();

shelve_dict.close()
end_timer("Shelved dictionary")
print("Dictionary line number:",line)
```

The timings obtained (on a test machine) were:

```
Brute_force : 1.014 seconds
Brute_force line number: 45400
Index       : 0.312 seconds
Index line number: 45400
In          : 0.296 seconds
Dictionary  : 0.016 seconds
Dictionary line number: 45400
Shelve creation: 38.517 seconds
Shelved dictionary: 0.468 seconds
Dictionary line number: 45400
```

The timings are not surprising, although Python 3.2 performance does not compare well with Python 2.7, which only takes round 1.7 seconds to create the same database. The Python 2 database file sizes are considerably smaller than those used by Python 3.

After creation, caching means that we are essentially dealing with an in-memory dictionary, nevertheless it is still slower than a conventional dictionary. The dataset is probably too small to meaningfully measure retrieval overheads. Since record sizes, and therefore the number of I/O transfers, will vary considerably between applications, a general measure with such a simple structure is not a reliable guide.

Functions

Objectives

To write and call our own user-written functions, and to continue practising Python.

Reference Material

Chapter 08 Functions. You might also wish to look at `os.times` in the online Python documentation.

Questions

1. Create a function which takes two arguments: a value and a list. The list should have a default of an empty list, for example:

   ```
   def myfunc(value, alist=[]):
   ```

   The function should append the value to the list then print the contents of the supplied list.

   Call this function several times with various different values, defaulting the list each time. Can you explain the output?

   Can you suggest a solution to the problem?

2. You may recall from a previous exercise that we had a couple of functions to time an operation. In this exercise you get to write those functions.

   The basis of the timing is to use **`os.times()`**. This returns a tuple describing the processor time used since the script started, and other items. We are only interested in the first two items in this tuple, the system time and user time. The total elapsed CPU time is the sum of these two items.

   Since the time given by **`os.times()`** is from some arbitrary moment, to start the timer we need to take the CPU time used so far and store it in a global variable. Do that in a function called **`start_timer()`**.

   In a function called **`end_timer()`** get the CPU time used, and subtract the time stored in the global variable to get the CPU time used between the two function calls. Display that CPU time, rounded to 3 decimal places, and a text message passed by the user as a parameter. The text should have a default value of "End time", and be a minimum width of 12 characters.

   Test your functions by having a long operation between the two calls. We suggest counting the number of lines in the 'words' file. For example:

   ```
   start_timer()
   lines = 0
   ```

```
for row in open ("words"):
    lines += 1

end_timer()
print ("Number of lines:",lines)
```

This code is already supplied in **Ex8.1.py**

3. Ah! We almost forgot. You did document your functions in the previous exercise, didn't you? Add docstrings to your functions, if you didn't already do it.

   To test: start IDLE and load your script (File/Open) then run it (<F5>). Then in the "Python Shell" windows type:
   >>> **help(start_timer)**
   >>> **help(end_timer)**

4. By now you should have seen the `country.txt` file in another exercise. It consists of lines of comma-separated values. If we wish to input these to a SQLite database using SQL, then these values (also called fields) must be slightly modified:
   a. Trailing white-space (including new-lines) must be removed.

   b. Any embedded single quote characters must be doubled. For example, `Cote d'Ivorie` becomes `Cote d''Ivorie`.

   c. All values must be enclosed in single quotes. For example, `Belgium,10445852,Brussels,737966,Europe` becomes: `'Belgium','10445852','Brussels','737966','Europe'`

   Write a Python script with a function to change a line into the correct format for insertion into an SQL statement, using the guidelines above.
   Call the function for each line in country.txt, and display the reformatted line.

   Hints:
   a. `rstrip()`
   b. `re.sub()`
   c. `lrow = []`
   ```
   for field in row.split(','):
       lrow.append("'" + field + "'")
   ```
   Then use `join()`

If time allows:

If you used the suggested '`for`' loop in the Hint, rewrite the code to use `map()` with a `lambda` function instead.

.

Solutions

Question 1

Our function:

```
def myfunc1(val, lista = []):
    lista.append(val)
    print("value of lista is:",lista)

myfunc1(42)
myfunc1(37)
myfunc1(99)
```

Output is:

```
value of lista is: [42]
value of lista is: [42, 37]
value of lista is: [42, 37, 99]
```

The problem is that the empty list is declared at the time of the function definition, which is at run-time. The default parameter is a reference to the empty list declared at that time. Subsequent default calls do not create a new list, they use the same one each time.

The normal Python idiom to solve this is to default to None instead:

```
def myfunc2(val, lista = None):
    if lista == None: lista = []
    lista.append(val)
    print("value of lista is:",lista)

myfunc2(42)
myfunc2(37)
myfunc2(99)
```

Output is:

```
value of lista is: [42]
value of lista is: [37]
value of lista is: [99]
```

Questions 2 & 3

```
import os
start_time = 0;

############################################################
# TIMER FUNCTIONS
def start_timer():
    """
    The start_timer() function marks the start of
    a timed interval, to be completed by end_timer().
    This function requires no parameters.
```

```
        """
        global start_time
        (utime,stime) = os.times()[0:2]
        start_time = utime+stime

    def end_timer(txt='End time'):
        """
        The end_timer() function completes a timed interval
        started by start_timer.  It prints an optional text
        message (default 'End time') followed by the CPU time
        used in seconds.
        This function has one optional parameter, the text to
        be displayed.
        """
        (utime,stime) = os.times()[0:2]
        end_time = utime+stime
        print("{0:<12}: {1:01.3f} seconds".
            format(txt,end_time-start_time))

    start_timer()
    lines = 0
    for row in open ("words"):
        lines += 1

    end_timer()
    print("Number of lines:",lines)
```

Question 4

```
import re

def prep_row (row):
    row = row.rstrip()
    row = re.sub(r"'",r"''",row)

    # Add quotes around each field
    lrow = []
    for field in row.split(','):
        lrow.append("'" + field + "'")

    return ",".join(lrow)


for row in open ("country.txt"):
    print(prep_row(row))
```

If time allows:

Lambda version:

```
lrow = list((map (lambda f: "'"+f+"'", row.split(','))))
```

Advanced Collections

Objectives

We will write a generator function using a custom built module, and use a dictionary comprehension.

Reference Material

Chapter 9 Advanced Collections, Chapter 8 Functions, and Chapter 5 Collections.

Questions

1. We have a small module written using the Python C API which iterates through the processes running on Windows. You might have to install this module, your instructor will give you guidance if that is the case. There is also a Linux version, but that one is pure Python.

   The advantage of this module is that it returns the parent process identifier (PPID) of each process, whereas conventional tools, like `tasklist`, do not.

   The module is called **`GetProcs`**, and the interfaces required for this exercise are:

   `GetFirstProc` Start iteration of running processes.

   `GetNextProc` Next iteration of running processes.

   Both functions take no parameters and return a tuple containing three items: (*integer process id*, *integer parent process id*, *string executable name*). False is returned at the end of the iteration.

   Write a Python program which has a generator function called **`iGetProcs`**. It should yield the same tuple as returned by the `GetProcs` interfaces. Make sure that the first process is not omitted! You will need to write test code for **`iGetProcs`**, we suggest using a `for` loop.

2. Using **`iGetProcs`** written in the previous exercise, use a dictionary comprehension to generate a dictionary where the keys are the process ids and the values are each a list containing the parent process id and executable name. The purpose of such a dictionary would be to lookup the parent process of any process id (you may extend this exercise to do that if you have time).

   Hints:

   This is simpler than the example shown in the course material - no `if` statement is required.

   If you get the error "ValueError: too many values to unpack" then consider unpacking. That is, prefix the tuple name with an asterisk.

Solutions

1. Here is our solution for `iGetProcs`:

```
import GetProcs

#############################################################

def iGetProcs():

    Retn = GetProcs.GetFirstProc()
    yield Retn

    while Retn:
        Retn = GetProcs.GetNextProc()
        if Retn:
            yield Retn

#############################################################

for proc in iGetProcs():
   print(proc)
```

2. The dictionary comprehension is fairly straightforward:

```
pids = {pid:value for pid,*value in iGetProcs()}
print(pids)
```

Modules and Packages

Objectives

To write and call our own user-written modules, and to continue practising Python.

Reference Material

Chapter 04 String Handling and Chapter 10 Modules and Packages.

Questions

1. In this exercise we will take two functions you wrote earlier and turn them into a module.

   The previous chapter included a question where you were asked to write two timing functions, **`start_timer()`** and **`end_timer()`**. If you did not complete that exercise don't worry, a sample solution is provided in **Ex10.1.py**. Use that file as a basis of this exercise, or your own solution if your wish.

   Create a module called **mytimer**, which contains these functions (and any other supporting variables). Test the module by importing it and calling the functions before and after a lengthy operation, as before.

   Note: there is a module called **timeit** in the Python Standard Library. If you look in the documentation you will find it is rather more complex then ours. On Windows there is also a module bundled with Python called **timer**. So do not use either of those module names.

2. Now test your module's docstring using IDLE. You did document your module, didn't you? If you did not, now is a good time.

   To test under IDLE, first **`import mytimer`**. Did that work? If IDLE did not find your module then maybe you should tell it where it is (hint: **`sys.path`**)? The easiest way to grab the path is to copy it from the Address bar in Windows Explorer and paste it into IDLE (use a "raw" string).

   Once you have managed to import the module, type:

   ```
   >>> help(mytimer)
   ```

3. Our module is not complete without some tests. Add a simple test to the docstring: call `start_timer()` immediately followed by `end_timer()`, so that the result is predictable. Do not forget to add the expected output. Then add the test for__ **`main`**__, with the call to **`doctest.testmod()`**.

   Test by running **`timer.py -v`** from the Windows command-line (cmd.exe).

If time allows…

Create a sub-directory called **mymodules**, and copy your timer.py module into it, but rename the file to **timer2.py**.

Add an empty **`__init__.py`** file to the sub-directory.

What modifications are required to your test code to use this package?

4. Write a module `printf.py` which provides functions similar to the C library routines sprintf, fprintf, and printf, using the 'old style' format syntax. See the slides after the summary of the "04 String Handling" chapter.

   Functions should be as follows:

   `sprintf(`*fmt*, **args*`)`
   Where *fmt* is a format string
   args is the argument list

   Returns a formatted string

   `fprintf(`*file*, *fmt*, **args*`)`
   Where *file* is a file object opened for write
   fmt is a format string
   args is the argument list

   Writes the formatted string to *file*

   `printf(`*fmt*, **args*`)`
   Where fmt is a format string
   args is the argument list

   Writes the formatted string to `sys.stdout`

Write **doctest** tests for your `printf` and `sprintf` functions. Note: omit "\n" from the format strings in your tests because doctest sees them as end-of-test.

Solutions

The test script looks like this:

```
import mytimer
mytimer.start_timer()
lines = 0
for row in open ("words"):
    lines += 1
mytimer.end_timer()
print ("Number of lines:",lines)
```

Here is our final module:

```
"""
This user written module contains a simple mechanism for
timing operations from Python.  It contains two
functions, start_timer(), which must be called first to
initialise the present time, and end_timer() which
calculates the elapsed CPU time and displays it.

>>> start_timer()
>>> end_timer()
End time    : 0.000 seconds
"""
import os
start_time = 0;

############################################################
# TIMER FUNCTIONS
def start_timer():
    """
    The start_timer() function marks the start of a timed
    interval, to be completed by end_timer().
    This function requires no parameters.
    """
    global start_time
    (utime,stime) = os.times()[0:2]
    start_time = utime+stime

def end_timer(txt='End time'):
    """
    The end_timer() function completes a timed interval
    started by start_timer.  It prints an optional text
    message (default 'End time') followed by the CPU time
    used in seconds.
    This function has one optional parameter, the text to
    be displayed.
    """
    (utime,stime) = os.times()[0:2]
    end_time = utime+stime
    print ("{0:<12}: {1:01.3f} seconds".
        format(txt,end_time-start_time))

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

If time allows…

The test script can be modified as follows:

```
import mymodules.mytimer2 as mytimer
```

That way we do not need to change the function call code.

Question 4

```
"""
    This module supplies functions sprintf, fprintf,
    and printf.

    >>> printf("%s","hello")
    hello
    >>> printf("%x",42)
    2a
    >>> printf("|%06.2f %-12s|",3.1426,"hello")
    |003.14 hello       |
    >>> var = sprintf("%X",3735928559)
    >>> print(var)
    DEADBEEF
"""
import sys

def sprintf(fmt, *args):
    rstr = fmt % args
    return rstr

def fprintf(file, fmt, *args):
    file.write(sprintf(fmt, *args))

def printf(fmt, *args):
    fprintf(sys.stdout,fmt,*args)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Classes and OOP

Objectives

To build a simple class using Python's object orientation, with particular emphasis on special functions and overloaded operators.

Reference Material

Chapter 11 Classes and OOP. Chapter 04 String Handling and Chapter 07 Data Storage and File Handling might also be useful. You will also need the Python online documentation.

Question 1

We are going to build a class for the records in `country.txt`. Each object will represent one country. The class will be called Country in **`Country.py`**.

The **`Country.py`** module has been started for you. It contains an index list giving the fields for each record in the `country.txt` file. You do not have to use this, there are many valid approaches. The comma-separated fields are as follows:

0 – Country name
1 – Population
2 – Capital city
3 – Population of the capital city
4 – Continental area
5 – Date of independence
6 – Currency
7 – Official religion (can be > 1)
8 – Language (can be > 1)

a) Take a look at the script **`user.py`**. This imports the Country module and reads the `country.txt` file. It creates a Country object for each record in the file, and stores it into a list called **`countries`**.

Each part of this question has a test in **`user.py`**, prefixed by a suitable comment. Currently all except the first one are commented out. Remove the # symbols at the front of the tests as you progress.

So, to get the **`user.py`** code to run as it is your first task is to implement a constructor (**`__init__`**) within the **`Country.py`** module.

We suggest that you keep it simple, and implement the object as a list of fields, created using `split()`.

b) The next task is to implement a **`print()`** method to print just the country name. Now, in **`user.py`**, remove the comments from the start of the second '**`for`**' loop and the first method call.

c) It would be easier for the user if the normal Python built-in `print()` could be used to print our object instead. Implement the **`__str__`** special method in **`Country.py`** to return the country name.

In **`user.py`** replace the call to the `print` method call with **`print(country)`**.

d) It is rather unwieldy to access the elements of the list in our methods. To make things simpler, implement two **getter methods**, one for the country name and another for the population field.

You can use the **`@property`** decorator – refer to the "**Properties and decorators**" slide in the course material.

Alter your **`__str__`** special method to use the getter method for name.

Remove the comments for this question from **`user.py`**, and the `print` statement for part c), then test.

e) The population totals for these countries are out of date as soon as the raw data is generated, so we need to be able to add or subtract numbers to these countries.

Write the special methods **`__add__`** and **`__sub__`** to add or subtract the required number to the country's population (we are ignoring the capital city for this exercise).

That sounds easy! Actually there is a sting in the tail of this one. You might have to alter the population field in the constructor to get this to work.

Hints:

when we read fields from a file they are *strings*.
we don't want to alter `self`, we want to alter (and return) a copy of `self`.

Uncomment the appropriate tests in **`user.py`** and run it. We are manipulating the population of Belgium in the test for no particular reason (apologies to Belgians – but we had to pick *somewhere*).

If time allows:

f) The **`user.py`** script holds a list of country objects, but we have no way of finding a particular country without printing it. We would like to use an **`index()`** method to search for a country name. We do not actually write an `index()` method for this but instead overload the **==** operator with the **`__eq__`** special method.

Again, uncomment the tests for this question and run them.

g) The fickle users have now decided that the **`__str__`** special function should output the population as well, and in a nice **format**. The country name should be left justified, with a minimum field wide of 32 characters, and be followed by a space, then the population right justified, zero padded, with a minimum width of 10 characters.

Remember **format**s? There were in the String Handling chapter.

Further, if time allows…

2. Construct a class called **File**. The constructor should take one parameter, a file name, and should create the file if it does not exist. Store the filename as an object attribute which is private to the *module*.

   The **File** class should also have a property called **size** which gives the size of the file in bytes (use `os.path.getsize()`).

3. Within the same module, add two further classes derived from **File**.

   a. TextFile

   Has a property called **contents**.
   The getter method returns the whole contents of the file as a string.
   The setter method appends text to the file.
   Both methods will need to open the file

   b. BinFile

   Has a property called **contents**.
   The getter method returns the whole contents of the file as a decoded string.
   The setter method appends data to the file. If the data is of class int (use `isinstance()` to check) then use `struct.pack()` to pack the integer into a binary format. If the data is of any other class then write it and an encoded string.
   Both methods will need to open the file as *binary*.

   Write suitable tests to write and read data to and from files, and check that the size method can be called from objects of the derived class.

Solutions

Question 1

As always there are many possible implementations, but here is ours:

```
import copy
class Country:
    index = {'name':0,'population':1,'capital':2,'citypop':3,
             'continent':4,'ind_date':5,'currency':6,
             'religion':7,'language':8}

    # Insert your code here
    # 1a) Implement a constructor
    def __init__(self, row):
        self.__attr = row.split(',')

        # 1e) Added to support + and -
        self.__attr[Country.index['population']] = \
            int(self._attr[Country.index['population']])

    # 1b) Implement a print method
    def print(self):
        print(self.__attr[Country.index['name']])

    # 1c) Overloaded stringification
    def __str__(self):
        #return self.__attr[Country.index['name']]
        # 1g) Formatting the output
        return "{0:<32} {1:>010}".
            format(self.__attr[Country.index['name']],
                self.__attr[Country.index['population']]))

    # Getter methods, using the @property decorator
    # See below for a non-decorator solution
    # 1d) Implement a getter method for country name
    @property
    def name(self):
        return self.__attr[Country.index['name']]

    @property
    def population(self):
        return int(self.__attr[Country.index['population']])
```

```
    # 1e) Overloaded + and -
    def __add__(self,amount):
        retn = copy.deepcopy(self)
        retn.__attr[Country.index['population']] += amount
        return retn

    def __sub__(self,amount):
        retn = copy.deepcopy(self)
        retn.__attr[Country.index['population']] -= amount
        return retn

    # If time allows:
    # 1f)  Overloaded == (for index search)
    def __eq__(self,key):
        return (key == self.name)
```

Non-decorator solution

Getter methods for name and population without using properties

```
    def name_get(self):
        return self.__attr[Country.index['name']]

    name = property(name_get)

    def population_get(self):
        return int(self.__attr[Country.index['population']])

    population = property(population_get)
```

Further, if time allows…

Questions 2 & 3 solutions

```
import os.path
import struct

class File:
    def __init__(self,filename):
        self._filename = filename

        # If the file does not exist, create it
        if not os.path.isfile(filename):
            open(filename,'w')

    @property
    def size(self):
        return os.path.getsize(self._filename)

# Text file
class TextFile(File):
    @property
    def contents(self):
        """ Return the contents of the file """
        return open(self._filename,'rt').read()

    @contents.setter
    def contents(self,value):
        """ Append to the file """
        if not value.endswith("\n"):
            value += "\n";
        open(self._filename,'at').write(value)

# Binary file
class BinFile(File):
    @property
    def contents(self):
        """ Return the contents of the file """
        value = open(self._filename,'rb').read()
        return value.decode()

    @contents.setter
    def contents(self,value):
        """ Append to the file """
        if isinstance(value,int):
            out = struct.pack('i',value)
            open(self._filename,'ab').write(out)
        else:

open(self._filename,'ab').write(value.encode())


if __name__ == '__main__':
    file1 = TextFile('file1.txt')
    file1.contents = 'hello'
    file1.contents = 'world'

    print(file1.contents)
```

```
print("Size of file1:",file1.size)

file2 = BinFile('file2.dat')

file2.contents = 42
file2.contents = 34
file2.contents = 'EOD'

print(file2.contents)
print("Size of file2:",file2.size)
```

Error Handling and Exceptions

Objectives

To try out Python exception handling within a module environment

Reference Material

Chapter 12 Error Handling and Exceptions.

Questions

1. Recall the **mytimer** module we worked on after Chapter 10 Modules and Packages. There were two functions, **start_timer()** and **end_timer()**, which should be called in that order. What if **end_timer()** was called without a **start_timer()** before it? We need to raise an exception in our timer module if that happens.

   Use your **mytimer.py**, or the one from the solutions directory. You will have to detect if **start_timer()** was called previously from the **end_timer()** function. We suggest that you reset your global time to zero in **end_timer()** after a successful run, and test that. Which exception would be appropriate to raise?

   Test it using **Ex12.py**.

2. Now, in **Ex12.py**, handle the error elegantly with an appropriate error message.

If time allows…

Ex12.py opens and reads the words file. What happens if that file does not exist? Handle that exception in and elegant manner as well.

3. In a previous optional exercise we created a class called **File**. If you did not complete that exercise then take file.py from the solution directory for 11 Classes and OOP.

   Handle an `IOError` in the constructor for **File**. Create a new attribute called **`_error`**, which should be False if the file is created successfully, but set to the exception arguments if there was an `IOError`.

   In the **`size`** method, return the file size (as before) if the object was created without an error, otherwise return **`None`**.

   Define a new property which returns the value of the **`_error`** attribute.

   Test your code. We suggest you create a directory and use that directory name for creating a file. Output an error message if there is an error with the file.

Solutions

1. Choosing which exception is not so easy. The nearest we could think of is SystemError. Given more time we might invent our own exception subclass. Raising the exception is fairly easy:

```
def end_timer(txt='End time'):
    """...
    """
    global start_time
    if start_time == 0:
        raise SystemError(
            "end_timer() called without a start_timer()")
    (utime,stime) = os.times()[0:2]
    end_time = utime+stime
    print ("{0:<12}: {1:01.3f} seconds".
        format(txt,end_time-start_time))

    start_time = 0
```

2. Detecting the error is also fairly straightforward:

```
try:
    mytimer.end_timer()
except SystemError as err:
    print ("end_timer error:",err, file=sys.stderr)
```

If time allows:

```
try:
    for row in open ("words"):
        lines += 1
except IOError as err:
    print ("Could not open:",
           err.filename, err.args[1],
           file=sys.stderr)
```

Question 3

```
import os.path
import struct

class File:
    def __init__(self,filename):
        self._filename = filename
        self._error = False

        # If the file does not exist, create it
        if not os.path.isfile(filename):
            try:
                open(filename,'w')
            except IOError as err:
                self._error = err.args

    @property
    def size(self):
        if self._error:
            return None
        else:
            return os.path.getsize(self._filename)

    @property
    def error(self):
        return self._error

# Text file
class TextFile(File):
    @property
    def contents(self):
        """ Return the contents of the file """
        return open(self._filename,'rt').read()

    @contents.setter
    def contents(self,value):
        """ Append to the file """
        if not value.endswith("\n"):
            value += "\n";
        open(self._filename,'at').write(value)

# Binary file
class BinFile(File):
    @property
    def contents(self):
        """ Return the contents of the file """
        value = open(self._filename,'rb').read()
        return value.decode()

    @contents.setter
    def contents(self,value):
        """ Append to the file """
        if isinstance(value,int):
            out = struct.pack('i',value)
            open(self._filename,'ab').write(out)
        else:
```

```
        open(self._filename,'ab').write(value.encode())


    if __name__ == '__main__':
        import sys

        # Test constructor error handling
        if not os.path.isdir:
            os.mkdir('Dummy')

        dummy = TextFile('Dummy')
        print("Size of Dummy:",dummy.size)

        if dummy.error:
            print("Dummy error:",dummy.error,file=sys.stderr)
        else:
            print("No error detected!",file=sys.stderr)
```

Multitasking

Objectives

To run external programs, in this case other Python scripts, using a variety of methods, first using the **subprocess** module and then using **multiprocessing**.

Reference Material

Chapter 13 Multitasking.

Question 1

In the **`labs`** or (on Linux) your home directory you will find a simple Python program, **client.py**, which lists files to STDOUT. The name of the file is specified at the command line, and if it cannot be read then an error is returned, using exit.

a) Now call the Python program **`client.py`** from another, passing a filename. If you can't think of a file to list, use the current program, or use the 'words' file.

   Output an error message if, for some reason, the **`client.py`** fails. Test this by:

   passing a non-existent file name
   calling a non-existent program

b) Modify the calling program to use a pipe and capture its output in a list. Print out the number of lines returned by the **`client.py`** program. Test as before.

Question 2

The purpose of this exercise is to experiment with different scenarios using the **multiprocessing** module. This is best demonstrated using a multi-core machine, so you might first like to check if that is the case. If not then the exercise is still valid, but not quite so interesting.

Note: IDLE, and some other IDEs, does not display output from the child processes run by the multiprocessing module. So run your code from the command-line.

Word prefixes are also called *stems*. We have written a program, **`stems.py`**, that reads the words file and generates the most popular stems of 2 to ***n*** characters long. It uses the **`mytimer`** module we created in a previous exercise, which you should make available.

Run the supplied **`stems.py`** program and note the time taken. You will note that no word exceeds 28 characters, so ***n*** could be 28, however we can

increase the value of ***n*** in order to obtain a longer runtime and demonstrate multiprocessing.

This time could be better used by splitting the task between cores. Using the **multitasking** module will require the stem search to be moved to a function. Make sure that all of the rest of the code is only executed in main (`if __name__ == '__main__':` test).

Scenarios:

a) ***n*** worker processes
This is where we split the task such that each stem length search runs in its own child process.

b) 2 worker processes ***n***/2 stem sizes each.
This assumes 2 CPU cores. It will require two processes to be launched explicitly, and each to be given a range of stem lengths to handle.

c) 2 worker processes using a queue.
This assumes 2 CPU cores. As in b), but instead of passing a range, pass the stem lengths through a queue. Make sure you have a protocol for the worker processes to detect that the queue has finished.

If time allows…

Question 3

Recall the SharePrices.py program in the 05 Collections exercises. Your job in this exercise is to make the program multi-threaded.

If you did not complete the previous exercise, then take a copy from the `solutions/05 Collections` directory.

The `SharePrices` dictionary itself needs to be changed. The value is now a list, the first element is the sequence (thread) number, and the second is the share price. Initialise `SharePrices` as follows:

```
SharePrices = {'Global Motors'   :['0',50],
               'Big Blue Inc.'   :['0',50],
               'Gates Software'  :['0',50],
               'Banana Computers':['0',50]}
```

You will need two functions:

`SetStockPrices`

Takes one argument – the sequence number.-
Loop continually, setting the sequence number, and the share price (as before), in `SharePrices`.

`ReadStockPrices`

No arguments are required to this function.
Loop continually printing out the details of the `SharePrices` dictionary every two seconds.

Note that each time we print out `SharePrices` the sequence number on each line should be the same for each member.

For example:

```
1 Banana Computers   $01.30
1 Global Motors      $02.14
1 Big Blue Inc.      $01.08
1 Gates Software     $02.70

3 Banana Computers   $01.03
3 Global Motors      $09.89
3 Big Blue Inc.      $01.16
3 Gates Software     $01.11
```

is OK, but:

```
2 Banana Computers   $01.30
1 Global Motors      $02.14
3 Big Blue Inc.      $01.08
1 Gates Software     $02.70

1 Banana Computers   $01.03
3 Global Motors      $09.89
```

```
2 Big Blue Inc.      $01.16
3 Gates Software     $01.11
```

is not!

So you will have to apply a lock each time you want to read or write to `SharePrices`.

Run four threads for each function. The sequence number is passed into each `SetStockPrices` thread and should be between 0 and 3.

Question 4

Find the Python program `Fcopy.py`. It copies files from your machine to the Instructor's machine, timing the operation. On Linux, note that a shared folder should be setup.

Run this program first to get a benchmark timing.

a. Write a multi-threaded version, using Queues. Have two worker threads (you can experiment with other number of threads if you wish) which actually do the copy, the main thread will pass filenames to these threads using a queue.

   After the copy has been done, each worker thread should pass the new filename to an additional thread that will delete the file, using a second queue.

   Finally, don't forget to place a marker (like False) onto the queue to indicate the end of the list, and wait (join) for all threads to complete.

   Did the multithreaded version run quicker?

b. Convert your multithreaded program to use the multiprocessing module. Does that run quicker?

Solutions

Question 1

```
from subprocess import *
import os
import sys
(a)
proc = Popen([sys.executable, 'client.py', 'words'])
proc.wait()
print ("Child exited with",proc.returncode)

(b)
proc = Popen([sys.executable, 'client.py', 'words'],
             stdout=PIPE, stderr=PIPE)
(output, error) = proc.communicate()

if error != None:
    print ("error:", error.decode())

print ("output:", output.decode())
```

Question 2

Base time on a machine with dual-core 2.66Gz CPU: Process : 1.125 seconds

Time for scenario a) (n processes) Process : 71.641 seconds

b) Process : 4.969 seconds

c) Process : 4.875 seconds

```
a)
import mytimer
from multiprocessing import Process
#################################################################
def stem_search(stems, stem_size):
    best_stem = ""
    best_count = 0
    for (stem,count) in stems.items():
       if stem_size == len(stem) and count > best_count:
           best_stem  = stem
           best_count = count
    if best_stem:
        print ("Most popular stem of size",stem_size,"is:",
               best_stem,"(occurs",best_count,"times)")

#################################################################
if __name__ == '__main__':
    mytimer.start_timer()
```

```
    stems = {}
    for row in open ("words"):
        for count in range(1,len(row)):
            stem = row[0:count]
            if stem in stems:
                stems[stem] += 1
            else:
                stems[stem] = 1
    mytimer.end_timer('Load')

    # Process the stems
    mytimer.start_timer()
    n = 30
    for stem_size in range(2,n+1):
        proc = Process(target=stem_search,
                       args=(stems,stem_size))
        proc.start()
        processes.append(proc)
    for proc in processes:
        proc.join()
    mytimer.end_timer('Process')
```

b)

```
import mytimer
from multiprocessing import Process
#################################################################
def stem_search(stems, start, end):
    for stem_size in range(start,end):
        best_stem = ""
        best_count = 0
        for (stem,count) in stems.items():
            if stem_size == len(stem) and count > best_count:
                best_stem  = stem
                best_count = count

        if best_stem:
            print ("Most popular stem of size",
                   stem_size,"is:",
                   best_stem,"(occurs",best_count,"times)")
```

```
################################################################
if __name__ == '__main__':
    mytimer.start_timer()
    stems = {}
    for row in open ("words"):
        for count in range(1,len(row)):
            stem = row[0:count]
            if stem in stems:
                stems[stem] += 1
            else:
                stems[stem] = 1
    mytimer.end_timer('Load')
    # Process the stems
    mytimer.start_timer()
    n = 30
    proc1 = Process(target=stem_search,
                    args=(stems,2,int(n/2)+1))
    proc1.start()
    proc2 = Process(target=stem_search,
                    args=(stems,int(n/2)+1,n+1))
    proc2.start()
    proc1.join()
    proc2.join()
    mytimer.end_timer('Process')

c)
import mytimer
from multiprocessing import Process, Queue
################################################################
def stem_search(stems, queue):
    stem_size = 1
    while stem_size > 0:
        stem_size = queue.get()
        best_stem = ""
        best_count = 0

        for (stem,count) in stems.items():
            if stem_size == len(stem) and count > best_count:
```

```
                best_stem  = stem
                best_count = count
        if best_stem:
            print ("Most popular stem of size",
                   stem_size,"is:",
                   best_stem,"(occurs",best_count,"times)")
################################################################
if __name__ == '__main__':
    mytimer.start_timer()
    stems = {}
    for row in open ("words"):
        for count in range(1,len(row)):
            stem = row[0:count]
            if stem in stems:
                stems[stem] += 1
            else:
                stems[stem] = 1
    mytimer.end_timer('Load')
    mytimer.start_timer()
    n = 30
    queue = Queue()
    proc1 = Process(target=stem_search, args=(stems,queue))
    proc2 = Process(target=stem_search, args=(stems,queue))
    proc1.start()
    proc2.start()
    for stem_size in range(2,n):
        queue.put(stem_size)
    queue.put(0)
    queue.put(0)
    proc1.join()
    proc2.join()
    mytimer.end_timer('Process')
```

If time allows…

Question 3

```
from threading import Thread
from threading import Lock
import time
import random

SharePrices = {'Global Motors'   :['0',50],
               'Big Blue Inc.'   :['0',50],
               'Gates Software'  :['0',50],
               'Banana Computers':['0',50]}

csSharePrices = Lock()

############################################################

def SetStockPrices(seq):
    # Updates stock prices with random price changes
    global SharePrices

    while True:
        csSharePrices.acquire()    # TODO
        for key,sp in SharePrices.items():
            SharePrices[key][0] = seq
            SharePrices[key][1] = max(1.0,
                sp[1] * ( 1 + ((random.random() –
                0.5)/0.5) * 0.05))

        csSharePrices.release()    # TODO

############################################################

def ReadStockPrices():

    global SharePrices

    while True:

        csSharePrices.acquire()    # TODO
        for key,sp in SharePrices.items():
            print("{} {:<18s} ${:05.2f}".\
                      format(sp[0],key,sp[1]))
        print()

        csSharePrices.release()    # TODO
        time.sleep(2)

############################################################
```

```
if __name__ == '__main__':

    tids = []

    # start share price update thread
    for i in range(0,4):
        th_set =
            Thread(target=SetStockPrices,args=str(i))
        th_set.start()

    # Wait for request from client
    for i in range(0,4):

        th_st = Thread( target=ReadStockPrices )
        th_st.start()
        tids.append(th_st)

    for tid in tids:
        tid.join()
```

Question 4

Here is our multithreaded version (without the timing routines), which actually runs slightly slower than the single threaded version.

```
import platform
import os.path
import glob
from threading import Thread
from queue import Queue

############################################################

def RemoveThread(*args):
    TargetDir,queue = args

    while True:
        FName = queue.get()
        if not FName: break
        os.remove(TargetDir + FName)


def WorkerThread(*args):

    TargetDir,queue,rqueue = args

    while True:

        FName = queue.get()
        if not FName: break

        Data = open(FName,'rb').read()

        FName = os.path.basename(FName)
        fh = open(TargetDir + FName,'wb')
        fh.write(Data)
```

```
            fh.close()

            rqueue.put(FName)

    ############################################################

    OperSys,Host = platform.uname()[:2]
    Source = './Bitmaps/*'

    if OperSys == 'Windows':
        TargetDir = '\\\\INSTRUCTOR\\Shared\\' + Host + '\\'
    else:
        TargetDir = '/mnt/hgfs/\\\\INSTRUCTOR/' + Host + '/'

    if not os.path.isdir(TargetDir):
        os.mkdir(TargetDir)

    start_timer()
    NumThreads = 2

    for i in range(0,10):

        print('Loop',i)
        queue  = Queue()
        rqueue = Queue()

        Tids = []

        for i in range(0,NumThreads):
            Tids.append(Thread(target=WorkerThread,
                        args=(TargetDir,queue,rqueue)))

        rth = Thread(target=RemoveThread,
                     args=(TargetDir,rqueue))

        for th in Tids:
            th.start()

        rth.start()

        for FName in glob.iglob(Source):
            queue.put(FName)

        for th in Tids:
            queue.put(False)

        for th in Tids:
            th.join()

        rqueue.put(False)
        rth.join()

    end_timer("Threaded:")
```

This is our multiprocessing version (without the timing routines), which runs considerably faster:

```
import platform
import os.path
import glob
from multiprocessing import Process, Queue
############################################################

def RemoveProcess(*args):
    TargetDir,queue = args

    while True:
        FName = queue.get()
        if not FName: break
        os.remove(TargetDir + FName)

def WorkerProcess(*args):

    TargetDir,queue,rqueue = args

    while True:

        FName = queue.get()
        if not FName: break

        Data = open(FName,'rb').read()

        FName = os.path.basename(FName)
        fh = open(TargetDir + FName,'wb')
        fh.write(Data)
        fh.close()

        rqueue.put(FName)

############################################################
if __name__ == '__main__':

    OperSys,Host = platform.uname()[:2]
        Source = './Bitmaps'

    if not os.path.isdir(Source):
        sys.exit("Unable to access "+Source)

    Source = Source + '/*'
    if OperSys == 'Windows':
        TargetDir =
            '\\\\INSTRUCTOR\\Shared\\' + Host + '\\'
    else:
        TargetDir =
            '/mnt/hgfs/\\\\INSTRUCTOR/' + Host + '/'

    if not os.path.isdir(TargetDir):
        os.mkdir(TargetDir)

    start_timer()
    NumProcs = 2
```

```
    for i in range(0,10):
        print('Loop',i)
        queue  = Queue()
        rqueue = Queue()

        Pids = []

        for i in range(0,NumProcs):
            Pids.append(Process(target=WorkerProcess,
                        args=(TargetDir,queue,rqueue)))

        rth = Process(target=RemoveProcess,
                      args=(TargetDir,rqueue))

        for th in Pids:
            th.start()

        rth.start()

        for FName in glob.iglob(Source):
            queue.put(FName)

        for th in Pids:
            queue.put(False)

        for th in Pids:
            th.join()

        rqueue.put(False)
        rth.join()

    end_timer("Multiprocessing:")
```

The Python Standard Library

There are no exercises for this chapter.