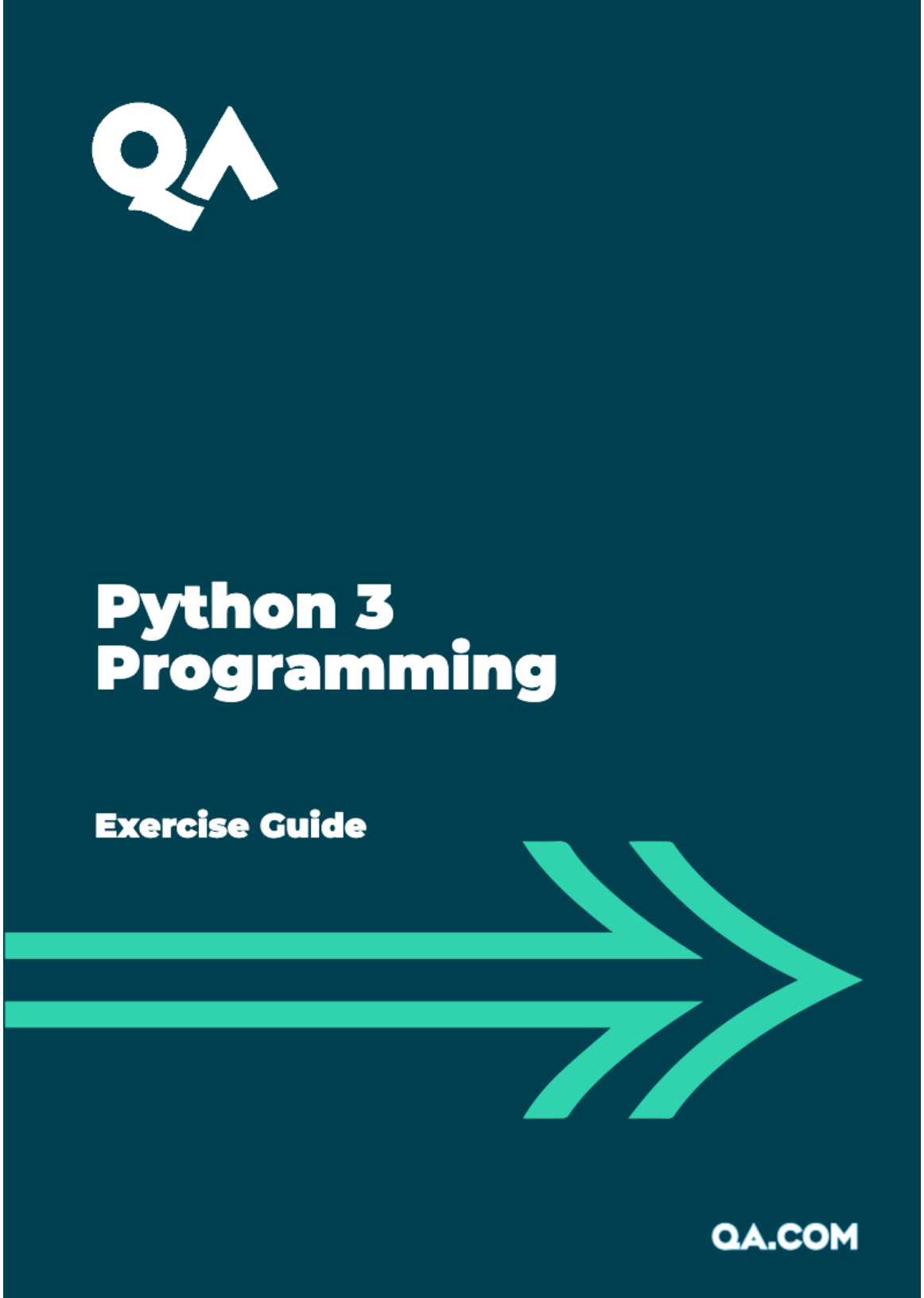
Python 3
Programming
Exercise Guide
QA.COM

Student setup for QAPYTH3

Download the latest version of Python from www.python.org. Make sure to check the box that asks if Python should be added to PATH.

Test by opening up a command window and running python.

Download PyCharm Community from the JetBrains site. Make sure it is the community edition, the professional edition requires a license.

Check your installation by:

- **File -> New -> Project**, choose a directory to store your files in e.g., PythonCourse. Wait until the progress bars bottom right have completed.
- **File->New->Directory** make a directory e.g., Ch01. Right click on this directory and **New->Python** file.
- Call it hello. In the text editor type: print**('Hello Python').**

Right click in your editor screen and select Run hello. You should see your output in the lower pane of PyCharm.

Downloading the QA slides, labs, and solution files: go **https://www.mimeo.digital/** - create an account. Log in and navigate to Archive, click on Redeem a key and enter the key that QA have given you for this course.

NOTE: Some corporations restrict or prohibit the downloading or emailing of zip files, if you have a personal laptop and it is in conformance with company policies to do so, use that for the course.

Introduction to exercises

The practical exercises are designed to supplement the technical training material in this course. Some 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 online 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 exercises.

Many of the exercises will require you to write complete programs, although these will be quite 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**. An OS X version is in development.

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**.

On OSX: As with Linux, the files should be in your home directory. Python 3 should be called as python3, and the #! line is **#!/usr/bin/python3**. Idle is started by calling idle3 from the command-line, but you might have an app launcher installed. Your instructor should be able to clarify.

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.

Exercise 1 – Introduction to Python 3

Objective

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

Questions

1. Which version of Python is being used?
2. Using IDLE (or equivalent), 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 PyCharm.

   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 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 being.

If time allows...

4. The card only displays on screen for 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>).

Question 1

Which version of Python is being used?

$ **python -V**

Python 3.10.5 (depending on what you have installed)

Question 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 PyCharm.

b) from IDLE.

Our solution has this code in it:

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

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

Question 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)
```

Exercise 2 – Fundamental Variables

Objective

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

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'll 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
```

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 fuel_requirement. 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 be 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: ")

# Display the value of var in upper case.
print(var.upper())
# Display the number of characters in var.
print(len(var))
```

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 in metres.

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?
fuel_per_lap = 2.25
laps = 45
fuel_requirement = laps * fuel_per_lap

# Typically, a car will carry an extra 50% for contingency.
fuel = fuel_requirement * 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.

q_lap_time = 80.45

# Theoretical initial lap time.
t_lap_time = q_lap_time - (0.35/10) * 5

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

lap_one_time = t_lap_time + ((fuel/10) * 0.35)
print("Lap one time:", lap_one_time, "seconds")
```

Exercise 3 – Flow Control

Objective

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

Questions

1. 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 three (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.

Optional extension

Passwords, and PINs, would not normally be displayed (*echoed*) to the screen for security reasons. So, now we will add the functionality to hide the characters typed. That could be a lot of work, but one of the advantages of using a language like Python is that "there's a module for it".

You'll need to **import** a module called **getpass**, which is part of the standard library.

Instead of **input** use **getpass.getpass**, in the same place in the program, with the same parameters.

Note: You will have to run your program in pycharm or VSCode. You may also have to update the **Edit Configuration Templates** to 'Emulate terminal in output Console' for the getpass module to work correctly in PyCharm.

2. 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...

3. 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: '))
```

4. 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 the dictionary os.environ with key HOMEPATH on Windows and 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.

Solutions

Question 1

There are several valid ways to write this code. Here's one solution:

```
import sys

PIN = '0138'
LIMIT = 4

for tries in range(1, LIMIT):
   supplied_pin = input('Enter your PIN: ')
   if supplied_pin == PIN:
      print('Well done, you remembered it!')
      print('... and after only', tries, 'attempts')
      break
# Note the else: is indented with the for loop, not the if!
else:
   print('You had', tries, 'tries and failed!')
```

Note: We used **uppercase** as a convention for constants, and we took advantage of the **else** on a **for** loop that is *not* executed on a **break**.

Optional extension to question 1

Using **getpass**, which is part of the standard library:

```
import sys
import getpass

PIN = '0138'
LIMIT = 4

for tries in range(1, LIMIT):
    supplied_pin = getpass.getpass('Enter your PIN: ')
    if supplied_pin == PIN:
        print('Well done, you remembered it!')
        print('... and after only' , tries, 'attempts')
        break
# Note the else: is indented with the for loop, not the if!
else:
    print('You had', tries, 'tries and failed!')
```

Why didn't we use **getpass** in the main question? Because making the input invisible makes debugging more difficult.

Question 2

Here's one simple solution using the range function:

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

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

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

Question 3

Here's our solution to test for leap years:

```
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 4

Here is our solution:

```
import sys
import glob
import os

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

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

# 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')
```

Exercise 4 – String Handling

Objective

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

Questions

1. Open the script **sep.py** in a text editor. You'll see a string defined called 'Belgium'. Add code to print:
    1. A line of hyphens the same length as the Belgium string, followed by...
    2. the string with the comma separators replaced by colons ':'., followed by...
    3. the population of Belgium (the second field), **plus** the population of the capital city (the forth field).

       Hint: The answer should be 11183818.
    4. A line of hyphens the same length as the Belgium string.

If time allows...

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

   MessierNumber CommonName ObjectType Constellation

   Note that many have no common name. Read the file using a **for** loop:

   ```
   for line in open('messier.txt', encoding='latin_1'):
    if not line: break
      # The text is in the variable named '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 fourth field).

Hint: The answer should be 11183818.

If you did this:

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

then you would've 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)
```

If time allows...

Question 2

```
for line in open('messier.txt'):
   if not line: break
   if line.startswith('M'):
      # Slice each field
      mes_num = line[:6].rstrip()
      com_name = line[6:40].rstrip()
      if not com_name: com_name = 'no name'
      obj_type = line[40:64].rstrip()
      const = line[64:].rstrip()
print(f"|{mes_num}|{com_name}|{obj_type}|{const}|")
```

Exercise 5 – Collections

Objective

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

Questions

1. What's wrong with this?

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

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

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

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

Prints 5

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

Prints 1

3. Write a Python script called Ex5_3.py that will generate and display 6 unique random lottery numbers between 1 and 50. Think about which Python data structure is best suited to store the numbers! Use the Python help() function to find out which function to use from the Python standard library called **random**.

4. 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_4.py in the **labs** directory (or your home directory if 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 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 users with their machine in any order. Print each user/machine pair on a separate line.

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

Solutions

1. What's 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's 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 function then reported the number of items in that tuple, which is one.

3. The name of the function to use is random.randint() and the Pythonic solution is store the numbers in a set. Objects stored in sets are unique.

   ```
   import random

   lotto = set()        # Create an empty set.

   while len(lotto) < 6:
       num = random.randint(1, 50)
       lotto.add(num)     # Add new number to set.


   print("Lottery numbers = ", lotto)
   ```

4. There are several solutions, here's 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))
```

Exercise 6 – Regular Expressions

Objective

To become familiar with some of Python's regular expression tools and continue practise with Python syntax and string handling.

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', 'r').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 += 1
    else:
        invalid += 1

infile.close()

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

If time allows:

Here's the additional code:

```
# Part c
valid_dict = {}
valid = open('validpc.txt', 'r').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 += 1
        # Part c
        area_district = m.group(1)   # Or alternatively m.groups(1)[0]

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

Exercise 7 – Data Storage and File Handling

Objective

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

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".
   - The /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'.

   On OSX, the file has unused ports marked as 'Unassigned'. Therefore, we have an additional requirement - ignore all lines that start with the comment delimiter '#'.

   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 integer.

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 named 'country.p'.

   Notice the size of the file compared 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.

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 an integer.

```
import sys
import re

if sys.platform == 'win32':
    file = r'C:\WINDOWS\system32\drivers\etc\services'
else:
    file = '/etc/services'

ports = set()

for line in open(file, 'r'):
    m = re.search(r'(\d+)/(udp|tcp)', line)
    if m:
        port = int(m.group(1)) # Or m.groups()[0]
        if port > 200: break
        ports.add(port)

# Subtract used port numbers from full set of ports
print(set(range(1, 201)) - ports)
```

Questions 2 & 3

```
import pickle
import gzip
import shelve

# Using a compressed pickle.
country_dict = {}
for line in open('country.txt', 'r'):
   name, *row = line.split(',')
   country_dict[name] = row

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

# Using a 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()
```

Exercise 8 – Functions

Objective

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

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 my_func(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 values, defaulting the list each time. Can you explain the output?

   Can you suggest a solution to the problem?

2. 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(my_func)**

3. 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 program 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 my_func1(val, lista=[]):
    lista.append(val)
    print("value of lista is:", lista)
    return

my_func1(42)
my_func1(37)
my_func1(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 my_func2(val, lista=None):
    if lista == None:
        lista = []
    lista.append(val)
    print("value of lista is:", lista)
    return

my_func2(42)
my_func2(37)
my_func2(99)
```

Output is:

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

Question 3

```
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", "r"):
    print(prep_row(row))
```

If time allows:

Lambda version:

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

Exercise 9 – Advanced Collections

Objective

We'll write a generator function and enhance it. This is an extension to the previous "Functions" chapter, as well as exercising our knowledge of generators and Python in general. If we have time, use a custom-built module and a dictionary comprehension.

Questions

1. By now, you should have an acquaintance with the built-in function **range()**. You will note that it only works on integers. This exercise is to write a version that handles floating-point numbers – float objects.

   * Please call your program **gen.py** – we will be using this in later exercises! *

   We won't be implementing everything that **range()** uses, we will make the first two parameters mandatory. We won't return a **range** object either, we will implement as a generator (a **range** object is a generator with extra features).

   Implement a version of **range()**called **frange()** with the following signature:

   frange(start, stop[, step])

   The default step should be 0.25.

   Note: Pay attention to the possibility of a mischievous user supplying a step of zero.

   Test with the following code:

```
print(list(frange(1.1, 3)))
print(list(frange(1, 3, 0.33)))
print(list(frange(1, 3, 1)))  # Should print [1.0, 2.0]
print(list(frange(3, 1)))     # Should print an empty list
print(list(frange(1, 3, 0)))  # Should print an empty list
print(list(frange(-1, -0.5, 0.1)))

for num in frange(3.142, 12):
    print(f"{num:05.2f}")
```

   Finally:

   print(frange(1,2))

   should show something like this:

   <generator object frange at 0x.....>

2. Enhance the **frange** function implemented in the previous exercise. The **range** function allows a single argument to be supplied that signifies the end of the sequence, the start then defaults to zero and the step defaults as before. Implement this in your **frange** function.

   Test with something like this:

   ```
   one = list(frange(0, 3.5, 0.25))
   two = list(frange(3.5))
   if one == two:
      print("Defaults worked!")
   else:
      print("Oops!  Defaults did not work")
      print("one:", one)
      print("two:", two)
   ```

3. This exercise is a further refinement of **frange**. It is the nature of floating-point numbers that inaccuracies appear, and you probably noticed some in your results. The inaccuracies are so serious that, as it stands, the function is not robust enough for a production environment.

   There are several solutions; one is to use the **decimal** module from the standard library. For that, we need to not only convert our function arguments to objects of the Decimal class but convert the result back to a float when we yield.

   The Decimal class constructor takes an integer or a string – this gives it the required precision. So, we need to convert our input parameters, for example:

   step = decimal.Decimal(str(step))

   Don't forget to import the **decimal** module and to yield a float. You should find the test results a little more sensible.

Solutions

Here are our versions of these exercises, remember that yours can be different to these, but still correct. If in doubt, ask your instructor.

1. Here's the first try at the **frange()** function (note do not use this in production code!):

```
def frange(start, stop, step=0.25):
    curr = float(start)
    while curr < stop:
        yield curr
        curr += step
```

2. This implements the enhancement to accept a single parameter:

```
def frange(start, stop=None, step=0.25):
    if stop is None:
        stop = start
        curr = 0.0
    else:
        curr = float(start)

    while curr < stop:
        yield curr
        curr += step
```

3. This is a more robust version of **frange**, using the **decimal** module:

```
import decimal

def frange(start, stop=None, step=0.25):
    step = decimal.Decimal(str(step))

    if stop is None:
        stop = decimal.Decimal(str(start))
        curr = decimal.Decimal(0)
    else:
        stop = decimal.Decimal(str(stop))
        curr = decimal.Decimal(str(start))


    if step != 0:
        while curr < stop:
            yield float(curr)
            curr += step
```

Exercise 10 – Modules and Packages

Objective

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

Questions

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

   In **Ex10_1.py** ,there are two timing functions, **start_timer()** and **end_timer()**.Use that file as a basis of this exercise, or your own solution if you 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 than 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.

   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).

Solutions

Here are our versions of these exercises, remember that yours can be different to these, but still correct. If in doubt, ask your instructor.

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 = None

# 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
   start_time = os.times()[:2]
   return

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.
    """
    end_time = os.times()[:2]
    print ("{0:<12}: {1:01.3f} seconds".
       format(txt, end_time - start_time))
    return

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

Exercise 11 – Classes and OO

Objective

To build a simple class using Python's object orientation. The first exercise will take you through the process of creating a simple file class step-by step. Most of the code is provided for you. The second exercise uses inheritance and is a little more challenging – you will have to figure out some of the code yourself.

Subsequent exercises are optional and are suitable only for experienced programmers.

Questions

1. This exercise will take you step-by-step into building a class based on simple file IO.

a. Create two files in the same directory:
myfile.py This will contain the class **MyFile.**
runfile.py This will import myfile and test the class.

In **myfile.py**, create a dummy class:

```
class MyFile:
  pass
```

In **runfile.py**, import the module, create an object, and print it:

```
from myfile import MyFile

filea = MyFile()
print(filea)
```

Execute **runfile.py**. Did you get the output you expected?

b. Now we will add a constructor to the MyFile class. It will require a file name to be input, which will be stored as an object attribute. In **myfile.py,** alter the class to say:

```
class MyFile:

  def __init__(self, filename):
    self._fname = filename
```

Now alter **runfile.py** to construct the object so that it passes the name of a text file. The file should exist, preferably in the current directory.

```
from myfile import MyFile

filea = MyFile('country.txt')
print(filea)
```

Execute **runfile.py**. The output should look the same as before.

c. Now we will make the print do a little more work, by providing a __str__ special function. In **myfile.py,** add a __str__function that will read the file into a string, and return it:

```
class MyFile(object):

  def __init__(self, filename):
    self._fname = filename

  def __str__(self):
    s = open(self._fname, 'r').read()
    return s
```

Test the code by executing **runfile.py**, the text file should be displayed. Note: this is only an exercise! It would be foolhardy to blindly read a file into a string, as it might be so large that it exceeds Python's memory allocation.

d. If we wanted to check the size of the file, we might wish to use the **len()** built in. For that, we can provide a __len__ method, to be added to **myfile.py**. This will require the **os.path** module, which we should import at the start of the module (outside the class).

We will also add a simple *getter* function to the class to return the filename, called get_fname. The class should now look something like this:

```
import os.path

class MyFile(object):

  def __init__(self, filename):
    self._fname = filename

  def __str__(self):
    s = open(self._fname).read()
    return s

  def __len__(self):
    return os.path.getsize(self._fname)

  def get_fname(self):
    return self._fname
```

Add a new test to **runfile.py**:

```
from myfile import Myfile

filea = Myfile('country.txt')
print(filea)

print(filea.get_fname(), "is", len(filea), "bytes")
```

Execute **runfile.py** and check the output.

2. Create a new module called **inherit.py**. In the new module, import the **MyFile** class from the **myfile** module. We are going to create two further classes derived from **MyFile**.

   Class **TextFile** will be for processing text files, and **BinFile** for processing binary files. In **inherit.py,** create these classes as being empty, inherited from **MyFile**. For example:

   ```
   class TextFile(MyFile):
   pass
   ```

   Do the same with the BinFile class.

   We can now add some tests. This time we will incorporate these tests in the **inherit.py** module itself, and run it as main – a common pattern.

   Add the "main" trap after the code for the new classes and do some simple tests. We have provided two *empty* files for these tests, **file1.txt** (text) and **file2.dat** (binary). Let's first check that the inheritance works, for example:

   ```
   if __name__ == '__main__':
      file1 = TextFile('file1.txt')
      print(file1, len(file1))

      file2 = BinFile('file2.dat')
      print(file2, len(file2))
   ```

   Assuming we have that working, we can move on to some more advanced operations using *properties*. Renew your acquaintance with properties by reviewing the slides if necessary.

   Start by removing the "pass" from both classes.

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 and should add a newline to the end of the text if none is present.
   Both methods will need to open the file, so we need to call get_fname(). For example:

```
    @property
    def contents(self):
    # Return the contents of the file
        return open(self.get_fname(), 'rt').read()

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

b. BinFile

Has a property called **contents**.

The getter method returns the whole contents of the file as a decoded string. It should be straightforward to write, so *have a go yourself*.

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 (so don't forget to import struct). If the data is of any other class, then write it and an encoded string. That is a little complicated, so here is the setter method:

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

The *str*.encode() is required because, being binary, we need to write a bytes object.

Both methods will need to open the file as *binary*.

Write suitable tests to write and read data to and from files. Here are some suggestions:

```
if __name__ == '__main__':
    file1 = TextFile('file1.txt')
    print(file1, len(file1))

    file1.contents = 'hello'
    file1.contents = 'world'

    print(file1.contents)
    print('Size of file1: ', len(file1))

    file2 = BinFile('file2.dat')
    print(file2, len(file2))

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

    print(file2.contents)
    print('Size of file2: ', len(file2))
```

If time allows...

3. In an earlier exercise, we created a generator called frange in a file called gen.py. The frange() function tried to emulate the standard built-in range(), but it falls well short of the task. The built-in range() function returns range objects on which you can perform various tasks.

   In this exercise, we will put our function into a class and create **Frange** objects. Note that the name of the class is capitalized to conform to PEP008, which, strictly speaking, range() does not.

   Take a copy of gen.py (or whatever file your frange() solution resides in) to create a new file called range.py.

   Of the stages that follow, consider stages 2 onwards to be optional.

   Stage 1

We can't just slap a class statement around the existing frange() function, we need to split it into two. We have the initialisation part and the generator part. Create a class called Frange and within it create two functions:

__init__ will take the same parameters as frange()did, plus the initial self. It will perform initialisation (conversion to Decimal) and add the stop, start, and step attributes.

__iter__ will only take the parameter self. It will consist of the iterator part of the old frange() function, that is the loop including the yield.

Alter the tests so that Frange() is called rather than frange().

Stage 2

Support the len() built-in function. In the constructor (__init__), we will add a length attribute. This can be done in the constructor because a range is immutable, and there are several operations which require it. The length of the range can be determined with:

self.len = math.ceil((self.stop - self.start)/self.step)

if self.len < 0: self.len = 0

Now add a __len__ method which returns the length attribute. Write a suitable test. Here is a suggestion:

```
if __name__ == '__main__':
    def printit(r):
        print(r, len(r), list(r))

    printit(Frange(1.1, 3))
    printit(Frange(1, 3, 0.33))
    printit(Frange(1, 3, 1))
    printit(Frange(3, 1))
    printit(Frange(-1, -0.5, 0.1))
    printit(Frange(1, 3, 0))
```

Stage 3

Right now, when we print an Frange object, we don't get anything which is useful. So, implement the __repr__ method into your Frange class. The string returned should be a command suitable for recreating the object, for example Frange(3.0, 1.0, 0.25). This assumes that start, stop, and step are attributes of self, so you might have to alter the constructor.

The class name should be obtained from self, not hard-coded (Hint: self has an attribute called __class__, which has an attribute called __name__).

Either write new tests or use those written previously.

Optional Extension:

Stage 4

Comparing two Frange objects could be done in by brute force, i.e., by comparing each element, but there is a simpler way. Simply compare the start, stop, and end positions used by the constructor. This assumes that they are attributes of self, which we added in the previous stage.

Add a __eq__ method with suitable tests. We could also add the others, __ne__, __gt__, and so on, but consider if these would be appropriate. If we just implement __eq__, do we need a __ne__? Try both == and != with only an __eq__ method.

If time allows...

4. 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) 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 is 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, but 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 reason (apologies to Belgians – but we had to pick *somewhere*).

f) The **user.py** script holds a list of Country objects, but we have no way of finding a 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. The population should be right justified, zero padded, with a minimum width of 10 characters.

Solutions

1. This is the final myfile.py:

```
import os.path

class MyFile:

  def __init__(self, filename):
    self._fname = filename

  def __str__(self):
    s = open(self._fname, 'r').read()
    return s

  def __len__(self):
    return os.path.getsize(self._fname)

  def get_fname(self):
    return self._fname
```

runfile.py:

```
from myfile import MyFile

filea = MyFile("country.txt")
print(filea)

print(filea.get_fname(), "is", len(filea), "bytes")
```

2. This is the complete inherit.py file:

```
import struct
from myfile import MyFile

# Text file.
class TextFile(MyFile):

  @property
  def contents(self):
    # Return the contents of the file.
    return open(self.get_fname(), 'rt').read()

  @contents.setter
  def contents(self, value):
    # Append to the file.
    if not value.endswith('\n'):
```

```
            value += '\n'
        open(self.get_fname(), 'at').write(value)
        return

# Binary file.
class BinFile(MyFile):

    @property
    def contents(self):
        # Return the contents of the file
        value = open(self.get_fname(), 'rb').read()
        return value

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

if __name__ == '__main__':
    file1 = TextFile('file1.txt')
    print(file1, len(file1))

    file1.contents = 'hello'
    file1.contents = 'world'

    print(file1.contents)
    print('Size of file1:', len(file1))

    file2 = BinFile('file2.dat')
    print(file2, len(file2))

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

    print(file2.contents)
    print('Size of file2:', len(file2))
```

3. Here is the complete Frange class. We have included a __getitem__ method for your interest (you were not asked, or expected, to provide this).

```
import math
import decimal
```

```
class Frange:

    def __init__(self, start, stop=None, step=0.25):
        self.step = decimal.Decimal(str(step))

        if stop is None:
            self.stop = decimal.Decimal(str(start))
            self.start = decimal.Decimal(0)
        else:
            self.stop = decimal.Decimal(str(stop))
            self.start = decimal.Decimal(str(start))

        # Calculate length.
        if self.step == 0:
            self.len = 0
        else:
            self.len = math.ceil((self.stop - self.start)/self.step)
            if self.len < 0: self.len = 0

    def __len__(self):
        return self.len

    def __getitem__(self, index):

        if index < 0:
            index = self.len + index

        if index >= self.len or index < 0:
            raise IndexError("index out of range")

        item = self.start + (index * self.step)
        return item

    def __repr__(self):
        retn = (f"{self.__class__.__name__:}({float(self.start):},"
                            + f" {float(self.stop):}, {float(self.step):})")
        return retn

    def __eq__(self, rhs):
        if (self.start == rhs.start and
             self.stop  == rhs.stop and
             self.step  == rhs.step):
            return True
        else:
            return False


        def __iter__(self):
```

```
        # Will return None on an empty list.
        if self.step != 0:
            curr = self.start
            while curr < self.stop:
                yield float(curr)
                curr += self.step
```

If time allows...

4. 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 printit(self):
        print(self._attr[Country.index['name']])
          return

    # 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...

2, 3

```
    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(afile):
    @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)
        return

# 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())
        return


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)
```

Exercise 12 – Error Handling and Exceptions

Objective

To try out Python exception handling within a module environment.

Questions

1. In 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 initialise and reset your global time to **None** in **end_timer()** after a successful run, and test that. We use **None** because zero is a valid time. 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 an elegant manner as well as creating a file. Output an error message if there is an error with the file.

Solutions

Here are our versions of these exercises, remember that yours can be different to these, but still correct. If in doubt, ask your instructor.

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 easy:

```
def end_timer(txt='End time'):
    """..."""
    global start_time
    if start_time is None:
        raise SystemError(
                "end_timer() called without a start_timer()")
    end_time = os.times()[:2]
    print("{0:<12}: {1:01.3f} seconds".
          format(txt, end_time - start_time))

    start_time = None
    return
```

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)
```

Exercise 13 – Testing

Objective

To test a function with both doctest and unittest

Questions

1. Create a new file called vat_calc.py. In this file, create a function called **vat_calc()** which accepts **amt** and **rate** as arguments. The function should take the amount and the VAT rate, and return the amount of VAT to be paid.

   Write a docstring which defines what the function does. Include a doctest with the values 100 and 20. These should return the value 20 (or 20.0 depending on the logic).

   Create a main function which outputs the following:

```
print(f"20% of 100 = {vat_Calc(100,20)}")
```

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

   Run the file to check that it has worked.

   Back in your function, write another test that uses 150.55 and 20 as the arguments. Add another line to your main function and retest (the value should be 30.11).

2. Create a new file called testVat_Calc.py. This file will be used to write the tests in an external file, which is far more likely in industry.

   Import the unittest first, then import your function from your first file.

   Unit tests are class based, so create a new class called **TestVat_Calc** which extends inherits from unittest.TestCase.

   Create a new def called **test_vat_calc(self)** and assertEquals that vat_Calc(100,20) is 20. Include a string to say what it should be at the end.

   When you check if **name** == **main**, call unittest.main().

   Run and check for output.

If time allows...

Install pytest.
Use this site How to write and report assertions in tests — pytest documentation to understand how to write code for pytest. Re-write your code to work with pytest.

Solutions

Here are our versions of these exercises, remember that yours can be different to these, but still correct. If in doubt, ask your instructor.

1.

```
def vat_Calc(amt, rate):
    """ Returns the amount of VAT to pay based on the amt and rate
    >>> vat_Calc(100, 20)
    20.0

    >>> vat_Calc(150.55, 20)
    30.11
    """
    return (amt / 100) * rate


def main():
    print(f"20% of 100 = {vat_Calc(100,20)}")
    print(f"20% of 150.55 = {vat_Calc(150.55, 20)}")
    return None

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

2.

```
import unittest
from vat_calc import vat_Calc

class Testvat_Calc(unittest.TestCase):
    def test_vat_Calc(self):
        self.assertEquals(vat_Calc(100,20), 20, "Should be 20.0")


if __name__ == '__main__':
    unittest.main()
```

If time allows...

```
def vat_calc(amt, rate):
    return (amt / 100) * rate


def test_vat_calc():
  assert vat_calc(100,20) == 20.0
```

In the terminal focused on the folder with your files:

```
pytest vat_calc.py
```

Exercise 14 – Multitasking & Asyncio

Objective

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

Questions

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.

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*** to obtain a longer runtime and demonstrate multiprocessing.

This time could be better used by splitting the task between cores. Using the **multiprocessing** module will require the stem search to be moved to a function. Make sure that all 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.

Solutions

1.

```
import subprocess
import os
import sys

#(a)
proc = subprocess.run([sys.executable, 'client.py', 'words'])
print('Child exited with', proc.returncode)

#(b)
proc = subprocess.run([sys.executable, 'client.py', 'words'],
          stdout=subprocess.PIPE, stderr=subprocess.PIPE)

if proc.stderr != None:
   print('error:', proc.stderr.decode())

print('output:', proc.stdout.decode())
```

2. The timings will obviously vary depending on the machine:

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)')
   return



if __name__ == '__main__':
```

```
    mytimer.start_timer()
    stems = {}
    for row in open('words', 'r'):
        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)')
    return

if __name__ == '__main__':
    mytimer.start_timer()
    stems = {}
    for row in open('words', 'r'):
        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)')
          return

if __name__ == '__main__':
   mytimer.start_timer()
   stems = {}
   for row in open('words', 'r'):
      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')
```

Want to find out more?

QA.COM