Unit 15 Exercises – GUIs, Canvas & Exceptions

Aims and objectives

- Further work with file processing, dictionaries, problem-solving and Exception handling
- Use of a lower-level drawing API (the Canvas module)

This week's exercises

These exercises do not introduce any new concepts, but you will be solving a more complex problem. The Moodle Unit15 folder contains test data for the exercises.

Task 0 – Predicting the outcome of python code snippets (optional)

Go to http://142.93.32.146/index.php/289256?lang=en and work through the exercises there.

- these exercises are not part of the course.
- they are entirely optional, yet relevant if you want to improve your programming skills.
- the results are anonymous.
- a group of academics (Fionnuala Johnson, Stephen McQuistin, John O'Donnell and John Williamson) hope to publish anonymized findings based on the collected results.

The Problem – Interpreting a language for drawing pictures

The aim of this exercise is to implement a program to read, from a file, instructions for drawing a picture, and draw the picture (using the *Canvas* module). The result will be an *interpreter* for a simple *programming language* for the specialised area of drawing.

A simple example of an input file is the following.

```
position 50 10
line 50 0
line 0 50
line -50 0
line 0 -50
```

This sequence of commands should be interpreted as follows. `position 50 10` sets the *current drawing position* to the coordinates (50,10). `line 50 0` draws a line from the current drawing position to a new point at a distance 50 horizontally and 0 vertically; this point becomes the current drawing position. The other `line` commands work similarly, so that the program above draws a square. You will need to use the following function:

- `create_line(xStart, yStart, xEnd, yEnd)`. The parameters give the starting and ending x and y coordinates of the line.

Another command is `move`; like `line`, this has two parameters, but it just changes the current drawing position without drawing a line. The program

```
position 20 10
line 0 50
move 20 0
line 0 -50
```

draws two parallel lines. Also include the command `circle`, which has one parameter and draws a circle with the specified radius and centred on the current drawing position.

After implementing an interpreter for the simple language described above, you can try to extend the language with *function definitions* and *loops*. For example, the program

```
define square
line 20 0
line 0 20
line -20 0
line 0 -20
end
position 20 10
loop 4
square
move 50 0
end
```

defines the function `square` (the lines from `define` to `end`) and uses a loop (the lines from `loop` to `end`) to draw four squares, moving the position 50 horizontally after each one.

If you would like a challenge, **don't read the rest of the exercise sheet**; just design and implement the program. Start by thinking about data structures, a plan for the program, and ideas for splitting the program into functions. Include error reporting: if there is an error in any command in the file, produce an error message including the line number. If you succeed and want a further challenge, try to extend the drawing language so that functions can have parameters.

If you continue reading overleaf, the tasks will guide you through the exercise.

Task 1 – The commands `position`, `line`, `move` and `circle`

Start with a program to interpret files containing these four commands only. Remember the basic structure of a loop to read a file line by line, and don't forget to do something about the newline character at the end of each line. Ignore completely blank lines (just as they are ignored in a Python program). You will of course need to import the *Canvas* module; to draw circles you will need the function

```
create_oval(x1,y1,x2,y2)
```

where `(x1,y1)` and `(x2,y2)` are the coordinates of the opposite corners of a rectangle containing the oval.

If any line of the file contains an error (for example, an invalid command name or an incorrect number of parameters), your program should print an error message including the line number. Justify your choice of defensive programming or exception handling to deal with erroneous input. Test your program with the file *test1.txt*, which should draw two triangles, and with other input files of your own.

Task 2 – Function definitions

Extend your program so that it interprets function definitions and function calls. For example, look at the file *test2.txt*, which should draw two squares. You will need to think carefully about the following questions.

- When you find the command `define`, how will you read the lines in the definition of the function? They should not be executed immediately.
- When a function is defined, you need to store its definition and name, so that when it is called, you can find the definition. What kind of data structure is suitable for this?

To avoid repetition of code, you will probably find it useful to structure your program so that there is a function which interprets a single command, and a main loop which calls this function for each command in the file. Think carefully about the parameters of this function.

Task 3 – Loops

Extend your program so that it interprets loops. The file *test3.txt* should draw a sequence of parallel lines. It should be possible to call functions within a loop. Allowing nested loops and loops within function definitions is more difficult, so you can ignore that if you want to.

Task 4 – Functions with parameters (optional)

Extend your program so that functions can have parameters. The exact syntax is up to you.