Assignment 1, Part B: Winning the Game

(5%, due 11:59pm Monday, April 20th, start of Week 7)

Overview

This is the second part of a two-part assignment. This part is worth 5% of your final grade for IFB104. Part A which preceded it was worth 20%. This part is intended as a last-minute extension to the assignment, thereby testing the maintainability of your code from Part A and your ability to work under time presssure. If you have a neat, clear solution to Part A you will find completing Part B easy. For the whole assignment you will submit only one file, containing your combined solution to both Parts A and B, and you will receive one grade for the whole 25% assignment.

Motivation

One of the most common tasks in "Building IT Systems" is modifying some existing code. In practice, computer programs are written only once but are subsequently modified and extended many times during their operational lifetime. Code changes may be required in response to internal factors, such as the need to correct design flaws or coding errors, or external factors, such as changes in consumer requirements.

A common situation is where some new feature must be added to an existing program. This is the scenario simulated in this part of the assignment. This task requires you to modify your solution to Part A of the assignment by adding a new capability. It tests:

- Your ability to work under time pressure; and
- The quality and clarity of your code for Part A, because a well-written solution to Part A will make completing this part of the assignment easy.

Goal

In Part A of this assignment you were required to develop a program which could follow a randomly-generated list of coded moves to simulate a game called "Not Connect Four", but we did not specify how the winner of the game is chosen. In the real game "Connect Four" the winner is the first player to get four tokens in a row, either vertically, horizontally or diagonally. However, this rule is considered too hard to implement for the assignment[1], so our "Not Connect Four" game uses a simpler rule. The winner of "Not Connect Four" is the first player to get four of their tokens topmost on four separate columns.

In this part of the assignment you will modify your solution to Part A to identify the winner of the game. Specifically:

- You must stop drawing tokens as soon as a player has won the game (or the list of moves is exhausted if there is no winner); and
- You must indicate visually which player has won (or whether the game was a tie in which case all players are considered equal winners).

[1] To see why, consider how much information about the state of the game is needed to recognise that a game of Connect Four has been won …

To complete this additional task you must modify your `play_game` function from Part A. No additional Python template file is supplied for this part of the assignment. Also, as per Part A, your Part B solution must work for any randomly-generated list of moves that can be returned by the provided function `random_game`.

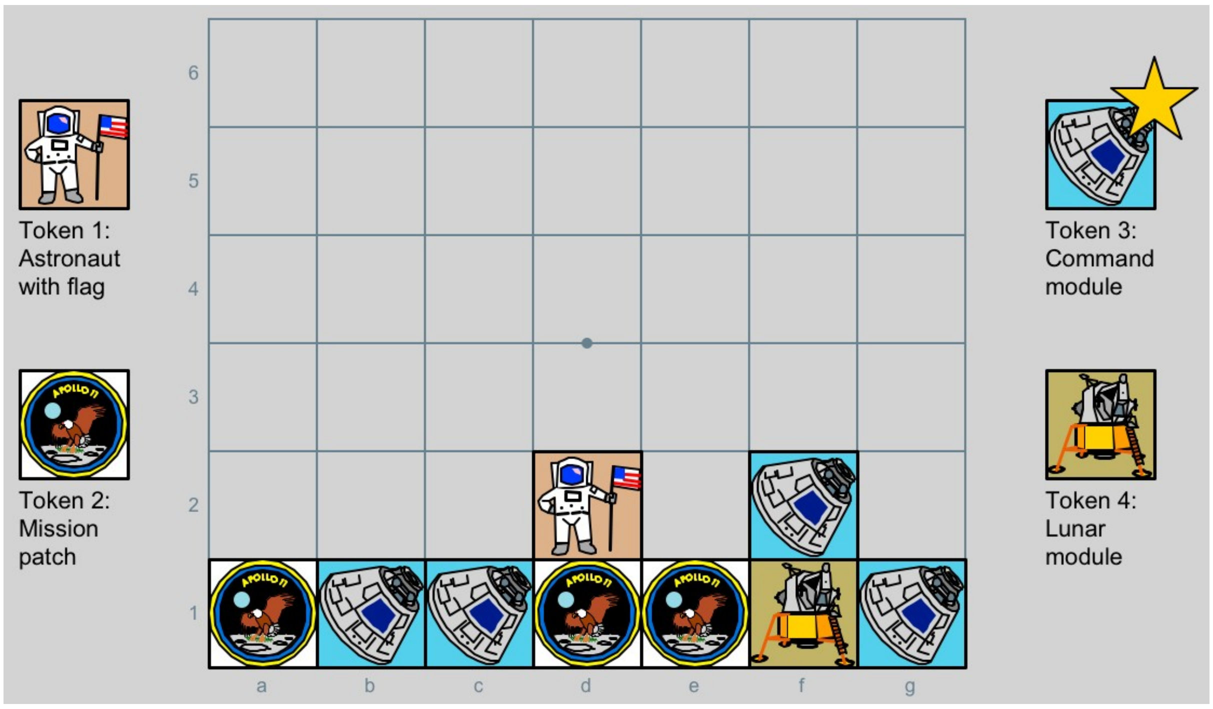
Illustrative example

To illustrate the requirement we continue our example from the Part A instructions. Recall that we developed a solution which drew four kinds of tokens celebrating the first moon landing. To meet the extended requirements of Part B we therefore developed additional code to recognise when a player has four tokens on the top of four different columns. As soon as this happens we stop drawing tokens, even if there are still some moves left in the list given to function `play_game`. For instance, consider the following short data set, generated by calling function `random_game`.

```
[['f', 4],
 ['d', 2],
 ['a', 2],
 ['c', 3],
 ['f', 3],
 ['e', 2],
 ['d', 1],
 ['g', 3],
 ['b', 3],
 ['b', 2],
 ['f', 3],
 ['b', 1],
 ['g', 2],
 ['g', 4]]
```

In this case 14 moves have been generated. The first is to drop a token of type 4 into column ‘`f`’. The second is token type 2 in column ‘`d`’, and so on. Importantly, however, tokens of type 3 are dropped into columns ‘`c`’, ‘`f`’, ‘`g`’ and ‘`b`’ early in the game. Although not immediately obvious from looking at the data set above, this results in tokens of type 3 occupying the topmost position of those four columns. This occurs after the ninth move. At this point the player using token type 3 has won the game because this is the first time the same type of token has been at the top of four columns. Our `random_game` function therefore stops drawing tokens at this point, because the game already has a winner, even though there are another five moves in the list above.

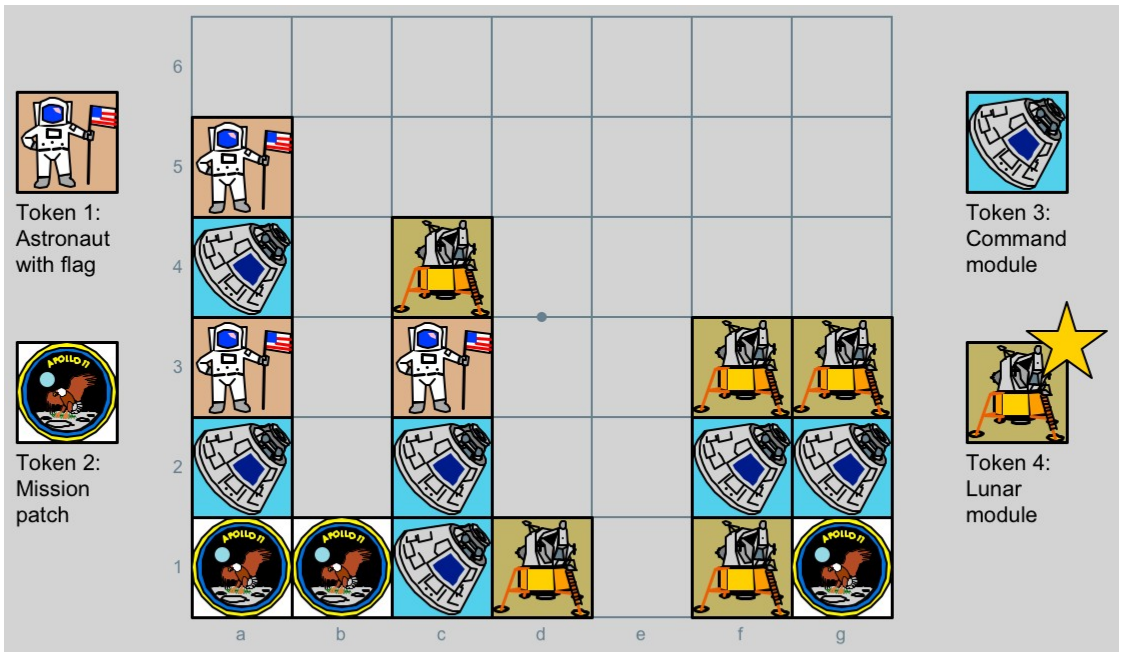
As well as stopping the game, the `random_game` function must also indicate visually who won. In our solution we have placed a gold star on the description of token 3, in this case the Apollo 11 Command Module. The resulting game drawn is as shown overleaf.



In the game above a player won after making only four moves, but this is not always the case. It's possible for a player to drop many tokens into the board without ever having four on top of different columns. As a longer game, consider the following sequence of moves.

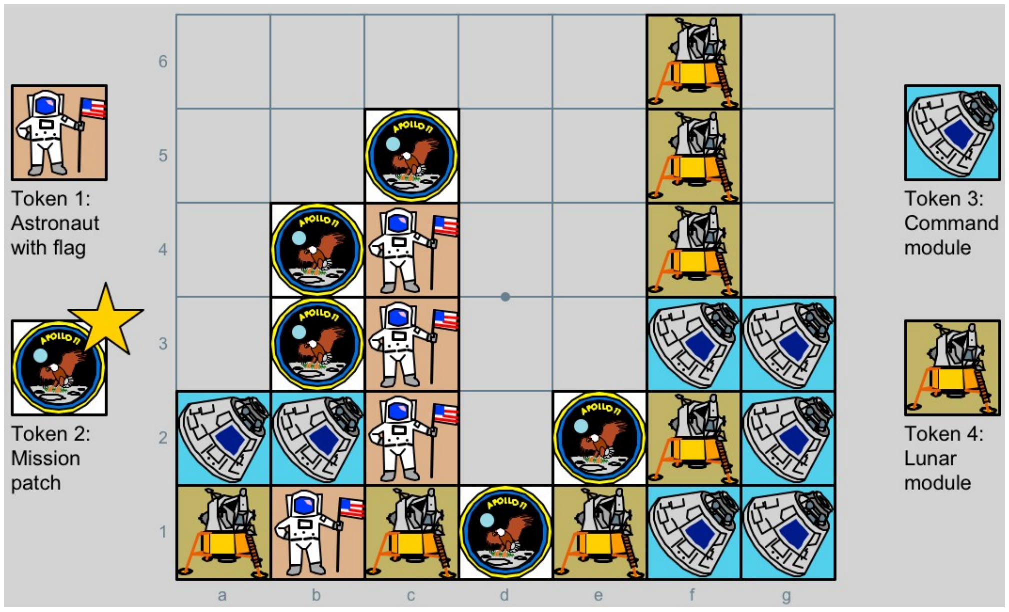
```
[['f', 4], ['f', 3],
 ['b', 2], ['c', 3],
 ['g', 2], ['a', 2],
 ['a', 3], ['a', 1],
 ['a', 3], ['a', 1],
 ['g', 3], ['f', 4],
 ['c', 3], ['g', 4],
 ['d', 4], ['c', 1],
 ['c', 4], ['a', 4],
 ['g', 3], ['d', 4]]
```

In this case 20 moves were generated. The resulting game is shown overleaf. It is eventually won after the 17th move when token 4, the Lunar Module, becomes the first token to appear at the top of four different columns. However, it took five moves by token 4 to achieve this winning position. By contrast, token 3, the Command Module, was played six times, but still didn't win because there was never a state in which this token was topmost in four columns.



Most games generated by function `random_game` involve a large number of moves and typically end with a player in a winning position. As another example, the following game has 38 moves but it isn't until the 23rd move that token 2 appears at the top of four columns.

```
[['c', 4], ['a', 4], ['c', 1], ['c', 1], ['g', 3], ['b', 1],
 ['e', 4], ['e', 2], ['b', 3], ['b', 2], ['b', 2], ['g', 3],
 ['f', 3], ['g', 3], ['f', 4], ['f', 3], ['d', 2], ['a', 3],
 ['f', 4], ['f', 4], ['c', 1], ['f', 4], ['c', 2], ['c', 3],
 ['g', 2], ['b', 3], ['g', 4], ['b', 3], ['a', 4], ['d', 4],
 ['e', 1], ['e', 2], ['d', 3], ['g', 4], ['d', 4], ['d', 3],
 ['d', 3], ['e', 4]]
```

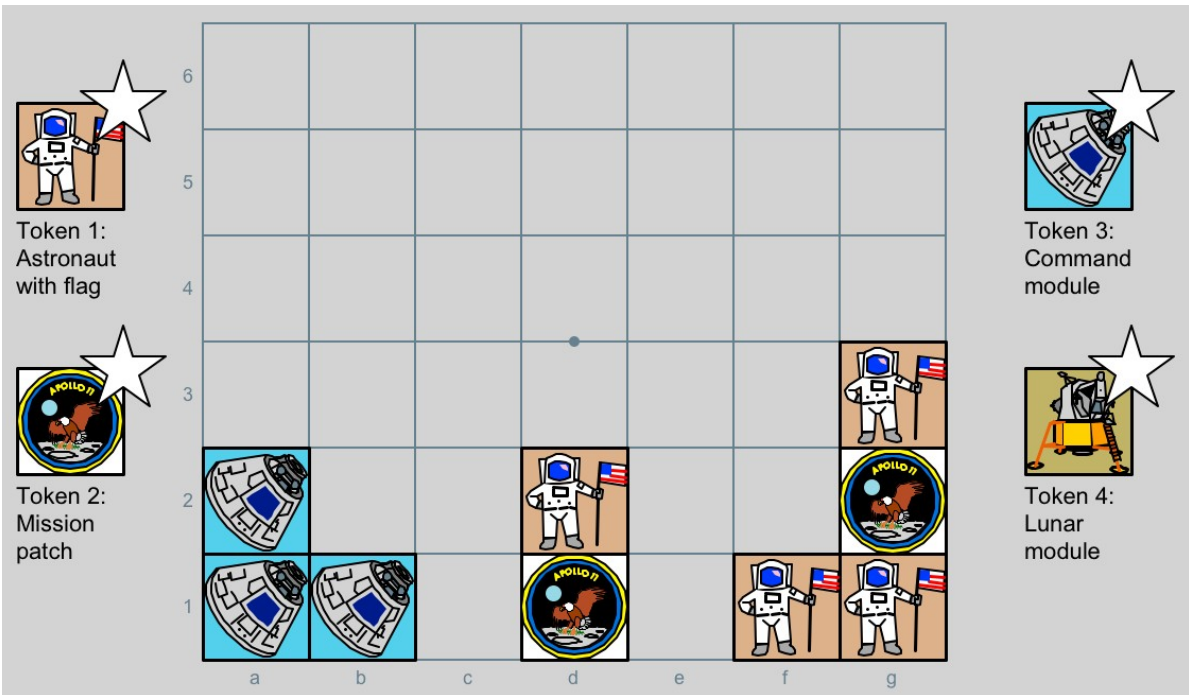


Nonetheless, it's possible that no token ever appears at the top of four columns at the same time. In this case the game will be played until the list of moves is exhausted. In such a situation we consider the game a tie and declare all four players equal winners.

This can happen, for instance, when only a small number of moves are generated by function `random_game`. Consider the following short sequence.

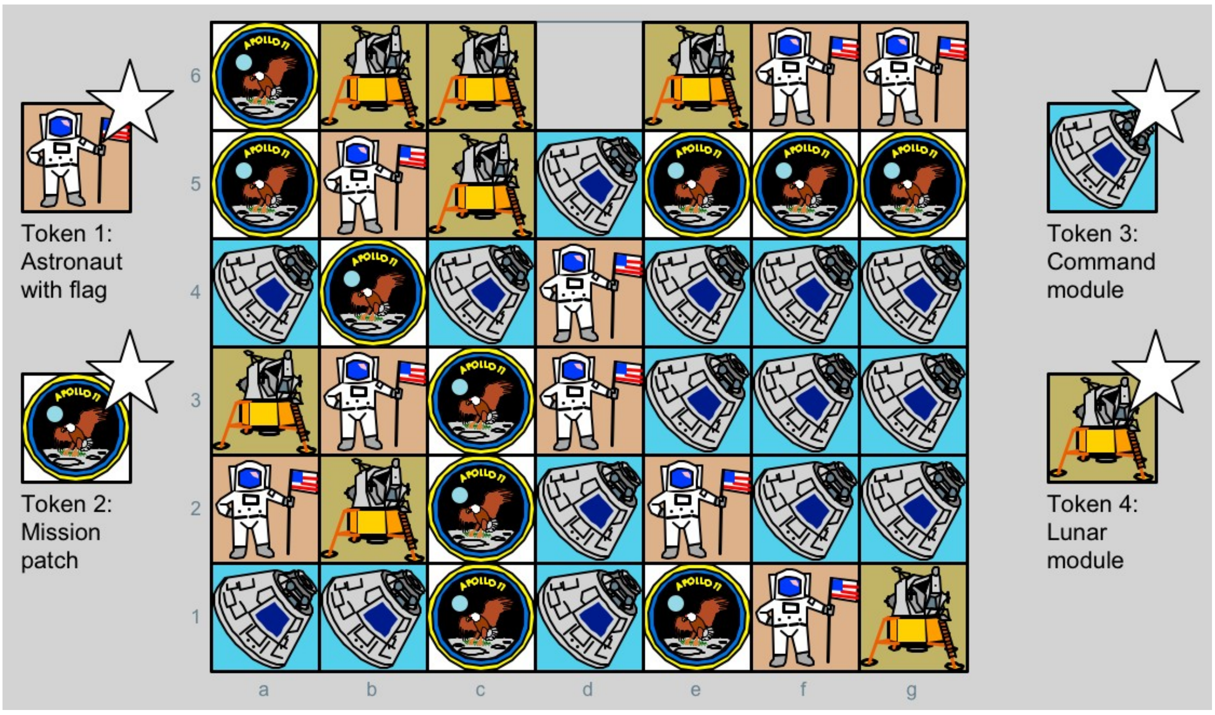
```
[['g', 1],
 ['g', 2],
 ['f', 1],
 ['a', 3],
 ['d', 2],
 ['b', 3],
 ['a', 3],
 ['g', 1],
 ['d', 1]]
```

In this case no token ever appears at the top of four columns before the list of moves ends. Token 1 is played four times, but never manages to appear on top of four different columns. Our `play_game` function therefore declares the game a tie. In this situation we award a silver star to all four players as shown below. (Player 4 was lucky enough to get an "equal-winner" silver star without even making a move!)



It's rare, but nonetheless possible, for the entire game board to fill with tokens without any player winning. The following sequence of 41 moves ends without any player achieving a winning situation, so the game is declared a tie after all 41 moves have been made as shown below.

```
[['g', 4], ['b', 3], ['g', 3], ['e', 2], ['g', 3], ['f', 1],
 ['g', 3], ['g', 2], ['f', 3], ['a', 3], ['g', 1], ['f', 3],
 ['a', 1], ['f', 3], ['a', 4], ['a', 3], ['c', 2], ['a', 2],
 ['d', 3], ['a', 2], ['e', 1], ['b', 4], ['e', 3], ['b', 1],
 ['d', 3], ['b', 2], ['b', 1], ['d', 1], ['b', 4], ['c', 2],
 ['c', 2], ['c', 3], ['e', 3], ['c', 4], ['f', 2], ['e', 2],
 ['d', 1], ['c', 4], ['e', 4], ['f', 1], ['d', 3]]
```



Resources provided

The Python template file accompanying the Part A instructions included a number of "fixed" data sets to help you debug your code during development. Although they can also be used for Part A, some of these data sets are specifically designed to help with Part B.

- Data sets `fixed_game_b0_0`, `fixed_game_b0_1` and `fixed_game_b0_2` all end in a tie.
- Data sets `fixed_game_b1_0`, `fixed_game_b1_1` and `fixed_game_b1_2` all reach states in which token 1 wins.
- Data sets `fixed_game_b2_0`, `fixed_game_b2_1` and `fixed_game_b2_2` all reach states in which token 2 wins.

- Data sets `fixed_game_b3_0`, `fixed_game_b3_1` and `fixed_game_b3_2` all reach states in which token 3 wins.
- Data sets `fixed_game_b4_0`, `fixed_game_b4_1` and `fixed_game_b4_2` all reach states in which token 4 wins.

Requirements and marking guide

To complete this task you are required to extend your Part A `not_connect_4.py` file by adding code so that it stops the game as soon as there is a winner, if any, and clearly indicates who won the game, including the possibility of a tie.

Your submitted solution for both Parts A and B will consist of a *single Python file*. Your Part B extension must satisfy the following criteria. Marks available are as shown.

1 **The game stops as soon as someone wins (3%)**. As explained above, your `play_game` function must draw tokens as per the supplied list of moves until either the list is exhausted or the first time a single player has tokens at the top of four different columns. Tokens must not continue to be drawn after a player wins to achieve marks for Part B of the assignment.

2 **The winner is clearly indicated (2%).** Once the game ends your program must indicate visually which player or players won. If a single player wins then it must be clear which player it was. If the game is tied then it must be made clear that all four players are declared equal winners. Any reasonable visual representation of the outcome is acceptable as long as it is easy to understand and highlights the correct player or players.

You must complete the assignment using basic Turtle graphics and maths functions only. You may not import any additional modules or files into your program other than those already included in the given `not_connect_4.py` template. In particular, you may not use any image files in your solution.

Development hints

- It should be possible to complete this task merely by *adding* code to your existing solution, with little or no change to the code you have already completed.
- If you are unable to complete the whole task, just submit whatever part you can get working. You will receive *partial marks for incomplete solutions*. Try to ensure that your program runs when submitted, even if it is incomplete.

Portability

An important aspect of software development is to ensure that your solution will work correctly on all computing platforms (or at least as many as possible). For this reason you must **complete the assignment using standard Turtle graphics, random number and maths functions only**. You may not import any additional modules or files into your program other than those already imported by the given template file. In particular, you may not import any image files to help create your drawings or use non-standard image processing modules such as `Pillow`.

Security warning and plagiarism notice

This is an individual assessment item. All files submitted will be subjected to software plagiarism analysis using the MoSS system (http://theory.stanford.edu/~aiken/moss/). Serious violations of the university's policies regarding plagiarism will be forwarded to the Science and Engineering Faculty's Academic Misconduct Committee for formal prosecution.

As per QUT rules, you are not permitted to copy *or share* solutions to individual assessment items. In serious plagiarism cases SEF's Academic Misconduct Committee prosecutes both the copier and the original author equally. It is your responsibility to keep your solution secure. In particular, **you must not make your solution visible online via cloud-based code development platforms such as GitHub**. Note that free accounts for such platforms are usually public. If you wish to use such a resource, do so only if you are certain you have a *private* repository that cannot be seen by anyone else. For instance, university students can apply for a *free* private repository in GitHub to keep their assignments secure (https://education.github.com/pack). However, we recommend that the best way to avoid being prosecuted for plagiarism is to keep your work well away from the Internet and your fellow students!

Deliverable

You must develop your solution by completing and submitting the provided Python 3 file `not_connect_4.py` as follows.

1. Complete the "statement" at the beginning of the Python file to confirm that this is your own individual work by inserting your name and student number in the places indicated. *We will assume that submissions without a completed statement are not your own work!*
2. Complete your solution by developing Python code to replace the dummy `play_game` function. You must complete your solution using only the standard Python 3 modules already imported by the provided template. In particular, you must *not* use any Python modules that must be downloaded and installed separately because the markers will not have access to these modules. Furthermore, you may *not* import any image files into your solution; the entire image must be drawn using Turtle graphics drawing primitives only, as we did in our sample solution.
3. Submit *a single Python file* containing your solution for marking. Do *not* submit multiple files. Only a single file will be accepted, so you cannot accompany your solution with other files or pre-defined images. **Do not submit any other files! Submit only a single Python 3 file!**

Apart from working correctly your program code must be well-presented and easy to understand, thanks to (sparse) commenting that explains the *purpose* of significant code segments and *helpful* choices of variable and function names. *Professional presentation* of your code will be taken into account when marking this assignment.

If you are unable to solve the whole problem, submit whatever parts you can get working. You will receive *partial marks for incomplete solutions*.

How to submit your solution

A link is available on the IFB104 Blackboard site under *Assessment* for uploading your solution file before the deadline (11:59pm Monday, April 20th, start of Week 7). You can *submit as many drafts of your solution as you like*. You are strongly encouraged to *submit draft solutions* before the deadline as insurance against computer or network problems near the deadline. If you are unsure whether or not you have successfully uploaded your file, upload it again!

Students who encounter problems uploading their Python files to Blackboard should contact *HiQ's Technology Services* (http://qut.to/ithelp; askqut@qut.edu.au; 3138 2000) for assistance and advice. Teaching staff will *not* be available to answer email queries on the evening the assignment is due, and Technology Services offers limited support outside of business hours, so ensure that you have successfully uploaded at least one solution by close-of-business on Monday, April 20th.