CS 1410 Assignment 4: Platformer

In this assignment, you will write a platformer game, where the player runs and jumps to find her way through a game level.

The assignment is split into three parts, each with its own due date.

The first due date covers the first two classes that you must implement, namely `Tileset` and `Map`. You should complete them, integrate them into `platformer.py` (removing any references to `tilesetsol` and `mapsol`), and have a working game. Submit this code and pass it off by the first due date.

The second part covers the `Guy` class, which is more complicated than the first two. You should complete it, integrate it into `platformer.py` (changing the `import` statement to reference `guy` instead of `guysol`), and have a working game. Submit the completed game and pass it off by the second due date.

In the third part of the assignment you must complete extensions to your platformer game as described below. You should complete these extensions, submit the completed game, and pass it off by the third due date.

Installing pygame

To handle the graphics and user input, you will use the `pygame` library. It is already installed on the lab computers (for all operating systems), but if you want to install it on your own machine you can find the files and instructions here:

- http://pygame.org/download.shtml

To install it on your own Ubuntu Linux machine, install the package `python-pygame` using this command:

```
sudo apt-get install python-pygame
```

To install it on Mac OS, follow the directions on this page:

- http://web.mit.edu/6.090/www/pygame.html

This document will provide all of the information you need to use `pygame` for this assignment, but you may find it helpful to try out a `pygame` tutorial or use the references, all of which are available here:

- http://www.pygame.org/docs/

The most important `pygame` classes for this game are `Image`, which you will use to load the graphics file, and `Surface`, which represents an image in memory. The game window is a `Surface`, and you will use it to render everything on the screen. You will also use the `Rect` class to represent a rectangle.

Getting started

Download the starter kit here:

- platformer.zip If you are working on your own computer, note that this kit has files that will only work with 2.7 versions of python.

Unzip it and run `main.py` to try it out. The starter kit consists of the following files:

- `main.py`: this is the code that starts everything up and drives the main loop.

  Note that the complete source code for `main.py` is provided. You should not need to change anything in it.

- `game.py`: this contains the generic game related logic to create a game window, watch for key strokes, and update the graphics in the game window 60 times per second. It also knows to quit when the user hits the escape key.

  Note that the complete source code for `game.py` is provided. You should not need to change anything in it.

- `platformer.py`: this contains the game logic specific to a platformer game like the one you are creating. It knows about tilesets, maps, and players.

Note that the complete source code for `platformer.py` is provided. You will only make changes to use your `Tileset`, `Map`, and `Guy` classes as you complete them. If you choose extensions that require it, you might need to change the `game_logic` to allow more interactions with the user.

- `config.py`: this file controls some of the game parameters. You can change values in here to change how fast the player moves, etc. It is used by `main.py` and `guy.py`.
- `tiles.png`: this image file contains all of the game graphics, arranged in *tiles*. A tile is a small (32×32 in our case) image used as a building block to display a complete game image. The entire map is composed of tiles arranged in a grid, and the player is animated by alternating which of four different tiles represents the player at any given time. If you want to change the look of the game, you can do so by creating a new set of tiles. This is discussed in more detail later.
- `jumper.map`: the game map. This is a very simple map, and you should create a more interesting map to replace it. This is discussed in more detail later.
- `jumper.tmx`: the Tiled source file for the map. See the section on creating your own maps.
- `convert-map.py`: a program to convert Tiled maps (with extension `.tmx`) into maps the game can understand (with extension `.map`).
- `tilesetsol.pyc`: a compiled version of the tileset manager that you must write. To start out, you can use this compiled code, but eventually you will replace it with your own file called `tileset.py`. This file contains a class that picks out the tiles from `tiles.png` and makes them available to the game.
- `mapsol.pyc`: a compiled version of the map manager that you must write. To start out, you can use this compiled code, but eventually you will replace it with your own file called `map.py`. This file contains a class that loads map files, keeps track of them in memory, and draws portions of them in the main game window.
- `guysol.pyc`: a compiled version of the player manager that you must write. To start out, you can use this compiled code, but eventually you will replace it with your own file called `guy.py`. This file contains a class that represents the guy (or gal) on the screen controlled by the player. All of the guy's behavior is implemented in this class, along with the code to draw it on the screen.

These files represent everything you need to play the game and try everything out. Three of the files are compiled python code that you will replace as you work through the assignment. Your task is to re-implement the `Tileset`, `Map`, and `Guy` classes, in that order. You can use `main.py`, `game.py`, `platformer.py` and `config.py` without any modifications, though you are free to change them to customize your game. In particular, you should name your game and put the name in `config.py`.

As you read the specifications for what you need to write, make sure that you have a solid understanding of what each piece does. From the python interactive shell, you can `import` each `.pyc` file and create instances of the classes they implement. Try calling each method to see what it does. Also, try doing as many of the tasks as possible by hand before implementing them in your project source files.

`platformer.py` currently `import`s the `sol` version of each of the three principal classes. As you implement your own, you can change `platformer.py` to use yours instead. You can do this incrementally, for example using `mapsol` and `guysol` but changing over to your version of `tileset`. Do not start a new class until you have everything working in the classes you have already implemented.

Part I

For the first due date, your job is to implement the first two main classes that control the game.

Tileset

The first (and easiest) is the `Tileset` class. This class is implemented in the `tilesetsol.pyc` file that you are provided. You should put your implementation in a file called `tileset.py`, and gradually stop using `tilesetsol.pyc`. When your `tileset.py` file is working, you can delete `tilesetsol.pyc` (or just ignore it). It is only there to help you get started.

The `Tileset` class loads an image file from disk, and splits it into a series of tiles. Load `tiles.png` into a browser or other image viewer so you can see how it is arranged. It has three rows of eight tiles, each 32×32 pixels in size:

1. The first row are background tiles, which are not solid (the guy can run and jump through them).

2. The second row are solid tiles, which the guy must jump over or go around.
3. The third row are character tiles. The first two have the guy facing left, and the second two have him facing right. A sense of motion is created by alternating between the two tiles for each direction (one has a foot up in the air, the other has the foot on the ground).

   Note that the guy tiles have alpha transparency. This means that the background pixels are invisible. `pygame` knows how to deal with these, so when the guy is painted on top of a background tile, the background shows through. This makes it look like the guy has a complicated shape, when in reality he is just a square tile like everything else.

An instance of the `Tileset` class represents the complete set of tiles used in the game. Note the distinction between *tiles*, which are `pygame` objects that can be rendered directly, and *tile numbers*, which are just integers that identify tiles. The `Tileset` class is responsible for loading the set of tiles, providing information about the set of tiles, and mapping tile numbers to tiles.

It has the following methods:

- `__init__(self, filename, tile_size_x, tile_size_y)`: the constructor takes three parameters. The first is the name of the image file containing the tiles. The second and third indicate how big a single tile is in pixels (both are 32 in our case).

  `pygame` knows how to read and decode most popular image file types, including `.png` files like ours. After importing `pygame`, you can load a file using:

  ```
  image = pygame.image.load(filename)
  ```

  The resulting object has a few methods of interest:

  - `image.get_size()`: returns the size of the image in pixels as a pair. Try loading an image by hand in the interactive shell, and call this method to see the size.
  - `image.subsurface(r)`: returns a `pygame` *Surface*, which is an object that represents some area where graphics can be stored or displayed. The window you see when playing the game is another Surface. This is the main method you will use to extract a single tile from the complete image. It expects a `Rect` object detailing the size and location of the rectangle that should be extracted from the image. You can create one of these using:

    ```
    r = pygame.rect.Rect(x, y, xsize, ysize)
    ```

    where `x` and `y` are the coordinates of the upper-left corner, and `xsize` and `ysize` indicate how big the extracted rectangle should be. Note that all coordinates start at 0×0 in the upper-left corner, and get bigger as you move right and down.

    As before, try creating one of these by hand and using it to call `subsurface`.

    Testing this by hand is a little harder than for some of the other methods. Start by opening a window (copy the code from `game.py` that initializes the screen), and then use the screen's `blit` method to paint your test tile on the screen. To do this, call:

    ```
    screen.blit(tile, (x, y))
    pygame.display.flip()
    ```

    where *x* and *y* are the coordinates where the top-left corner of the tile should be displayed. The `flip()` function tells `pygame` to actually display whatever has been painted. It uses something called *double buffering* to make the animation smoother. In a nutshell, it does all painting off-screen, and then you flip buffers and it displays whatever you have painted. Then you draw the next frame off-screen and call `flip()` again to display your next set of changes. When testing, you need to do it each time you want to see what you are working on.

    Try doing this for a few of the tiles that you create by hand before moving on.

  The `Tileset` constructor has a few jobs. It should store the tile size in `self` for later access. It should load the image from disk. It should get the size of the image, and compute how many rows and columns of tiles it must contain (8 columns and 3 rows for now, but it should work with larger and smaller images as well). It should store these values in `self` for later reference as well. Finally, it should use `subsurface` to extract all of the tiles from the image and store them in `self` for later retrieval. They should be stored in a list, starting at the top-left and moving across the first row, then the second row, etc. The map will refer to tiles by this number, so the first background tile is number 0 for our example, the first solid tile is number 8, and the guy tiles start at number 16.

- `getTileSize(self)`: this method just returns the size of tiles in pixels as a pair (the same numbers that were given to the constructor).

- `getTileCoordsAt(self, x, y)`: this method takes coordinates on a map in pixels, and returns the same coordinates in tile blocks. For example, if it is given `(40,190)` as its *x* and *y* inputs, it would note that 40 pixels from the left edge of a grid of tiles would be somewhere in the second tile (tile number 1 when starting from 0), and 190 pixels from the top of a grid of pixels would be somewhere in the sixth tile (tile number 5).

  Note that this is simpler than it sounds. Just divide *x* by the `tile_size_x` that was passed to the constructor, and divide *y* by `tile_size_y`. Return the two values as a pair.

- `getTile(self, tilenumber)`: returns one of the tile objects by its number, as described above.

- `getBackgroundTile(self)`: returns the default background tile, which is just tile number 0. This is what gets displayed when the screen shows something off the edge of the map.

- `getFirstCharacterTileNumber(self)`: return the tile number where the character tiles begin, i.e., where the third row begins. This is to help the `Guy` class compute which tile should be used to animate the guy. Note: do not hard-code this as 16, since that will not work with other tile sets. Instead, calculate the number for the first tile in the third row.

- `isSolid(self, tilenumber)`: this method returns `True` if the given tile number is on the second row of tiles, i.e., the solid row. It returns `False` otherwise. Again, do not hard code the range of numbers that are solid. Instead, calculate where the second and third rows begin, and test if `tilenumber` is in that range.

Map

The second class is `Map`. An instance of this class represents a map, also called a "level" in the game. A map is just a rectangular grid of tiles.

Just as `Tileset` is implemented in `tilesetsol.pyc`, a starter version of `Map` is provided in the `mapsol.pyc` file. Your job is to implement your own version in `map.py`, which will replace `mapsol.pyc`. When you are finished, you will not need `mapsol.pyc` at all.

The `Map` class loads a map from disk and stores it. Load `jumper.map` into a text editor to see how map files are defined. The overall file is just a python list of rows, where each row is a list of tile numbers. You could create a new map in this format by hand, but there is an easier way descibed later.

The `Map` class has the following methods:

- `__init__(self, filename, tiles, screen, view_x, view_y)`: the constructor takes the name of the map file, a `Tileset` object, the screen (a `pygame` Surface where everything is rendered), and the size of the game window (in tile blocks, not pixels).

  The constructor should start by storing its arguments for later use (inside `self`), and then load the map file. To do this, read the entire file into a string (open it and use the file's `read()` method to load the whole thing). Then you can ask python to interpret the string as though it was python code, returning the result, by using the built-in function `eval`:

  ```
  self.map = eval(filecontents)
  ```

  After doing this, `self.map` will be a list of rows, where each row is a list of tile numbers. Don't forget to close the file.

  You should also count the number of rows (the length of the `self.map` list), and the number of columns (the length of one of those rows) and store those values for later use. You may assume that all maps will be rectangular.

- `getSize(self)`: this method returns the size of the map (in tile blocks, not pixels) as a pair, width first, then height.

- `getViewSize(self)`: this method returns the size of the screen view (in tile blocks, not pixels) as a pair, width first, then height.

- `getTileNumber(self, x, y)`: this method returns the tile number from a given position on the map. If the position is invalid (off one of the edges of the map, for example if *x* is negative), then return `None` instead.

- `getTileNumberAt(self, x, y)`: methods with names ending in `At` refer to pixel numbers, while those that do not refer to tile block numbers. This method returns the tile number at the given position. Note that you can use the

`getTileCoordsAt` method in the `Tileset` object to do the conversion for you, then call `self.getTileNumber`.

- `getTile(self, x, y)`: this method returns the actual tile at a given position, not just the tile number. It should call `getTileNumber` to get the tile number. If that method returns `None`, it should return the background tile as given by the `getBackgroundTile` method of the `Tileset` object. Otherwise, it should pass the number on to `getTile` (also part of the `Tileset` object) and return the result.

- `isSolidAt(self, x, y)`: this method retrieves the tile number at the given pixel coordinates, and returns `True` if it represents a solid tile. Use `isSolid` from the `Tileset` object to determine this. Note that any position off the map should be considered solid.

- `drawMap(self, left, top)`: this is the trickiest method. It draws the visible part of the map onto the screen (recall that the screen object was given to the `Map` constructor).

  The parameters give the position (in pixels) of the top-left pixel to be displayed. You may find it helpful to compute the tile position where that pixel falls (divide each coordinate by the size of a tile), and the *offset*, or how far that pixel is from the corner of the tile. Compute the offsets by taking the remainder after dividing by the size of a tile (use the `%` operator).

  Remember that to obtain the size of a particular tile, you can execute a call to your `tileset.getTileSize()` method.

  It is okay to render blocks that do not completely fit on the screen: `pygame` will simply clip the parts that do not fit in the window. However, you should not render the entire map each time, or it will be too slow.

  The simplest way to do it is to render a view that is one bigger in each direction than the actual screen is. The following pseudo-code will walk you through it:

```
(corner_x, corner_y) = (left / tile_size_x, top / tile_size_y)
(offset_x, offset_y) = (left % tile_size_x, top % tile_size_y)


loop over screen_y values from 0 to viewsize_y inclusive:
    map_y = screen_y + corner_y
    loop over screen_x values from 0 to viewsize_x inclusive:
        map_x = screen_x + corner_x
        get tile at position (map_x, map_y)
        render the tile at the following position:
            (screen_x * tile_size_x - offset_x,
             screen_y * tile_size_y - offset_y)
```

  To render the tile, use the `blit` method of the screen object (which was passed in to the `Map` constructor). Use `blit` like this:

```
screen.blit(tile, (left_position, top_position))
```

  This will draw `tile` (which is a `pygame` Surface) on `screen` (which is also a `pygame` Surface), positioning its top-left corner at the given coordinates. Note the parentheses around the coordinates. The top-left corner coordinates are what the pseudocode above gives.

Part II

For the second due date, your job is to implement the third main class that controls the game.

Guy

The third class is `Guy`. An instance of this class represents the player on the screen. It is responsible for managing motion and animation, detecting collisions, and rendering itself on the screen.

By now, you should be using your implementations of `Tileset` and `Map`. `platformer.py` should have "`import tileset`" instead of "`import tilesetsol as tileset`", and "`import map`" instead of "`import mapsol as map`".

Following this pattern, `Guy` is implemented in `guysol.pyc`. Your job is to implement your own version in `guy.py`, which will replace `guysol.pyc`. When you are finished, you will not need `guysol.pyc` at all.

The `Guy` class directly accesses the data in `config.py`, so it should `import` the module.

The `Guy` class tracks the following information at all times:

- Basic information about the environment. These are values that are created in `platformer` and passed in to `Guy`, including the `screen` object, the `Tileset` object, and the `Map` object.
- The current position of the guy on the map. This position is specified in pixels, and refers to the position of the top-left corner of the guy's tile (note that this ignores the `HOWSLIM` parameter in `config`—it is based on a standard 32×32 tile size). The coordinates it tracks are relative to the top-left corner of the map.
- The current velocity of the guy, in pixels per frame. This is stored as two values: *dx*, the horizontal distance the guy travels per frame, and *dy*, the vertical distance the guy travels per frame. These are `float` values, which means they can be fractions of a pixel.
- The direction the guy is currently facing (left or right). This can be represented as a boolean variable that is `True` if the guy is facing right, `False` otherwise.
- Whether or not the guy's foot is currently up. The guy is animated by alternating between two tiles, one with his foot up, and the other with it down. The class needs to track which one is currently being displayed. It also needs to track how many steps (or pixels) the guy has moved with his foot in the current position.

Most of the methods in the `Guy` class are concerned with manipulating these values. In addition, the guy knows how to ask the map to draw the current view (based on the guy's position), and to draw himself on the screen in the correct position.

The `Guy` class has the following methods:

- `__init__(self, tiles, map, screen, x, y)`: the constructor takes the `Tileset`, the `Map`, the screen, and the starting position of the guy. It should store all of these values for later use.

  The constructor also initializes all of the values listed above that the class must track. The guy starts out at rest (no movement), facing right with his foot down (and having taken zero steps with his foot in this position).

- `pushX(self, ddx)`: this method nudges the guy horizontally, affecting his current speed (not current position). Note that this method does not actually move the guy, it just accelerates his rate of motion.

  The speed is subject to some limits, which should be checked in this method. The guy cannot move faster than `config.MAXIMUM_RATE_X` in either direction. If this push would exceed that limit, set the speed to exactly that limit. Note that this applies in the positive and negative directions.

  In addition, if this push results in the guy having a positive rate of motion, you should record that the guy is facing right, and if it results in negative rate of motion, you should record that the guy is facing left.

- `pushY(self, ddy)`: this is similar to `pushX`, but in the vertical direction. The final rate of vertical motion is limited by `config.MAXIMUM_RATE_Y`. Unlike `pushX`, there is no need to consider whether the guy is facing up or down.

- `left(self)`: this is called each frame when the user is holding down the left arrow key. It should push the guy left (using `pushX`) the amount set in `config.ACCELERATION_PER_FRAME`.

- `right(self)`: this is the same as `left`, except that the user is holding down the right arrow key and the guy should be pushed to the right.

- `friction(self)`: this method applies the effect of friction in a given frame. It should check the current horizontal rate of motion, and if it is greater than zero it should push the guy left. If it is less than zero, then it should push the guy right (using the `pushX` method). The amount it should push should be the amount set in `config.FRICTION_PER_FRAME`. If that amount is too much (it would push the guy in the other direction), then it should push just enough to stop the guy instead.

- `gravity(self)`: this method applies the effect of gravity in each frame. It does so by pushing the guy down the amount specified in `config.GRAVITY_PER_FRAME`. It should always apply this push, without regard to the current rate of motion (unlike friction). Remember that down is increasing the value of `y`, and up is decreasing the value of `y`.

- `jump(self)`: this method makes the guy jump, but only if his feet are on the ground. It can test this by calling the `feetOnGround` method (defined below). You may wish to wait until you have implemented that method before writing this one.

  If the guy's feet are on the ground, implement a jump by pushing the guy vertically (using `pushY`) *twice* the maximum vertical motion rate (defined in `config.MAXIMUM_RATE_Y`). This will ensure that—no matter what his prior vertical motion rate—the guy will end up moving vertically as fast as possible.

- `roundUp(self, n)`: this is a helper method used in the method `move`, described below. It rounds its argument *n* up to the nearest `config.MINIMUM_PIXELS_PER_FRAME`. For example, if `MINIMUM_PIXELS_PER_FRAME` is 4, then we would get:
  - `roundUp(0.0)` returns 0
  - `roundUp(0.5)` returns 4
  - `roundUp(3.9)` returns 4
  - `roundUp(4.0)` returns 4
  - `roundUp(4.6)` returns 8

  `roundUp` should return an `int`, but it is given a `float` value. It can be implemented as follows:

```
to = config.MINIMUM_PIXELS_PER_FRAME
return ((int(math.ceil(n)) + to-1) / to) * to
```

- `move(self)`: this method applies motion for a single frame. This method must respond to collisions while moving the guy, and control when the guy's foot moves up or down.

  1. Start by calculating how many pixels the guy should move in each direction. For the *x* direction, track the absolute number of pixels the guy should move `abs(self.dx)`, and the sign of the direction, -1 for left and +1 for right. Use `roundUp` to round the absolute number of pixels up to an integer.

     Repeat the same calculations for the *y* direction. The sign should be -1 for up, and +1 for down.

     You should now have 4 numbers:

     - the rounded up absolute number of pixels for *x*
     - the sign of the *x* motion
     - the rounded up absolute number of pixels for *y*
     - the sign of the *y* motion

  2. Next, apply the motion in the horizontal (*x*) direction, one pixel at a time. Before moving the guy each pixel, use `collision` (described below) to test if the move would result in a collision. If so, this indicates the guy hit a wall. Do not move him, instead set the horizontal rate of motion to zero. If there is no collision, apply the single pixel of motion. Repeat this process until a collision occurs, or the absolute number of pixels for *x* as occurred.

     Remember the total number of pixels that the guy did move before hitting something, or completing all horizontal motion.

  3. Next, do the same for vertical motion. Apply it one pixel at a time, always checking in advance if the move would cause a collision. If not, apply it and continue, but if so, set the vertical motion rate to zero and stop trying to move vertically (no need to track how many vertical pixels were successfully traversed).

  4. Next, check if the guy's foot should be up or down. You should be tracking the total number of steps taken in the current state of foot motion (up or down).
     `config.PIXELS_PER_STEP` is the number of pixels that the guy should travel before changing the state of the foot.

     Retrieve the total number of steps taken in the current state from `self`. Add the actual number of horizontal steps taken above. Divide the result by (`2*config.PIXELS_PER_STEP`) and save the remainder. (Use the `%` operator.) Save the result as the new value of number of steps taken in the current foot cycle (`state`).

     If `steps` is in the range 0 ≤ `steps` < `perstep`, the guy's foot should be up. If it is in the range `perstep` ≤ `steps` < `2*perstep`, the guy's foot should be down.

  5. If the guy is not moving (`round(dx) == 0.0`), his foot should be down and `steps` is reset to zero. Note that this overrides anything done in the previous step.

  6. If the guy is in the air (his feet are not on the ground), his foot should be up and `steps` is reset to zero. Note that this overrides anything done in previous steps.

- `collision(self, x, y)`: this method returns `True` if moving the guy to the given location (given as pixel coordinates of his top-left corner) would result in him overlapping a solid tile. This is tested by asking if any of the four corners of the guy would overlap a solid tile.

  Start by computing the position of the guy's left and right sides, and his top and bottom:

- Left side: this is the *x* position passed in plus `config.HOWSLIM`, because the guy's coordinates are always tracked based on a full-sized tile, but the guy is actually narrower than that.
- Right side: this is the *x* position plus the width of a tile, minus one (because we want the guy's right side, not the first pixel to his right), minus `config.HOWSLIM`.
- Top: this is just the *y* position passed in.
- Bottom: this is the *y* position passed in plus the height of a tile minus one (we want the guy's bottom pixel, not the pixel just below him).

Once you have these numbers, return `True` if the map reports a solid tile at any of the corner locations, i.e., if the map's `isSolidAt` method returns `True` for any of the guy's corners. Otherwise, return `False`.

- `feetOnGround(self)`: this method tests if there is a solid tile immediately below the guy. This method works by asking the `collision` method if moving the guy down one pixel would cause a collision.
- `drawGuy(self)`: this method should draw the guy's tile on the screen. It must first decide which tile to use. Note that the first guy tile is the one returned by the `getFirstCharacterTileNumber` method of the `Tileset` class. The second guy tile is one past it, etc.
  - If the guy is facing left and has his foot up, use the first guy tile.
  - If the guy is facing left and has his foot down, use the second guy tile.
  - If the guy is facing right and has his foot down, use the third guy tile.
  - If the guy is facing right and has his foot up, use the fourth guy tile.

  To draw it on the screen, use the `blit` method of the `screen` object as described earlier. The *x* and *y* coordinates of the guy never change. Compute them using:

  ```
  x = (view_size_x - 1) / 2 * tile_size_x
  y = (view_size_y - 1) / 2 * tile_size_y
  ```

  where the view size is in tile blocks.
- `repaint(self)`: this method repaints the entire screen. It starts by calling the map's `drawMap` method. To compute which part of the map should be displayed, take the guy's position and subtract half the size of the view, i.e.:

  ```
  x = guy_position_x - ((view_size_x / 2) * tile_size_x)
  y = guy_position_y - ((view_size_y / 2) * tile_size_y)
  ```

  This is the position you should give to `drawMap`.

  After painting the background, draw the guy's tile by calling the `drawGuy` method.

Custom maps

To create your own custom maps, use the map editor included in the starter kit. The file is called `tiled.jar`. This file is all you need, but if you want the source code and everything else, go here:

- http://mapeditor.org/

This is a java program, and to run it you must have java installed:

- http://java.com/en/download/

You should be able to right-click on `tiled.jar` and run it with java, or from the command line you can type this to load it:

```
java -jar tiled.jar
```

Editing an existing map

This editor understands `.tmx` files, like the `jumper.tmx` file in your game directory. You can load it using the menus, or you can load it when you start the program:

```
java -jar tiled.jar jumper.tmx
```

At the bottom of the screen there is a window pane (drag it up so you can see it) with all of the tiles. Click on them and "paint" them onto the map. Ignore everything to do with layers and other features. When you save your changes, it should be into a `.tmx` format file. To convert a `.tmx` file into a `.map` file that your game can understand, use the `convert-map.py` script provided in the game directory:

```
./convert-map.py mymap.tmx mymap.map
```

Then you can launch the game with that map:

```
./main.py mymap.map
```

or you can change `config.py` to use it by default.

Creating a new map

If you are starting a new map from scratch in the editor, make sure the map type is "orthogonal", and make sure the tile size is 32×32 (or whatever size you have used if you are using a custom tile set). You can make your map any size.

To get access to the tiles, select "New tileset" from the "Tilesets" menu. Select the "Reference tileset image" box and browse to find the tile image file. Make sure the "Tile height" property is set correctly (32 for the default tiles). All other default options should be fine.

Changing the graphics

To change the tileset, just create a new image made up of tiles that are all the same size (you do not have to stick to 32×32). You can do this in Photoshop or any other image editing software. Collections of 32×32 tiles are available on the web; this site hosts a few such collections:

- http://www.purezc.com/index.php?page=tilesets

Note that changing the size of the tile file will change the numbering of the tiles, which will break existing maps. You should create your tile file first, then create maps using it. If you change the size of the tile file after creating a map, you may find that the map looks wrong and needs to be corrected.

If you create a large set of tiles, you may want to break from the system of putting a single type of tiles on each row, e.g., you may want many rows of solid tiles. In this case, you will need to change your `Tileset` code to understand the new system of organizing a tile image file.

Part III

There are many things you can do to make this a better game. Do not work on these enhancements until you have all of the basic functionality complete and working.

You must complete at least two items from the list below. If you would like to add a different enhancement, clear it with the instructor before working on it.

- Add multiple levels and a way to move between them. Perhaps arriving at a certain "doorway" location automatically transports you to a specific level. Or, you could make it so that going off the edge of the map takes you somewhere else.
- Create a new map in the aforementioned way. The map needs to be substantially different than the one given to you in the starter code. You could also use a different tileset. One online source of free tilesets is here:

  http://www.tekepon.net/fsm/modules/refmap/index.php?mode=vx-map

  (click on English in the upper-right corner)
- Create ways to die. Perhaps falling too far and hitting the ground, or landing on certain tiles (spikes), or running out

of time.

- Add things to be collected. “Coins” or other objects, or perhaps arriving at certain locations acts like gathering an object (you win when you have made it to all 4 towers).
- Other kinds of motion. Maybe you can climb walls, or jump extra high under certain conditions, or use a rope to get to a high ledge. Or you run at different speeds when holding down the `<shift>` key.
- Interact with the environment. Break through bricks, or fall through the floor with a certain command.
- Have different environments. In a water level, motion is slower, friction is less, and there is no (or weak) gravity. Maybe you float up instead of falling down. Or some surfaces are icy, and it is harder to speed up or slow down on them. Or the window size is smaller in an underground level, so you cannot see as far.
- Have other creatures. Friends or foes wandering around the game, which help you or hurt you when you run into them.
- Add background music and sound effects. Pygame has built-in support for this kind of thing, so it may be easier than it sounds.
- Add different animation effects. Use different tiles for jumping and falling, for motion and standing still, etc. Or have the environment animated: have some wall tiles that alternate between different images (perhaps every second or so), similar to how the guy is animated.

What to turn in

- For the first deadline, you should submit only your `tileset.py` and `map.py` files.
- For the second deadline, you just need to add `guy.py`, so that all three files are submitted.
- For the third deadline, you need to submit all files necessary to see your version of the platformer game. Please submit a `.zip` file containing everything needed to run the game. Include a `README` file that describes what you have done. If you do not include this description, your work will not be evaluated.