5 Instructions

All of you need to install Pyglet. For that you can open a Terminal (command line tool / bash shell) and type:

```
pip install pyglet
```

When applications get more evolved, there is often a need to split up the single tasks inside the application. In our case we use the Model-Controller-View concept. The Model is responsible for the game logic and all the calculations (That would be the server side in a multiplayer application). The View is responsible for the graphics (client side). It should not do any calculations itself. The Controller communicates between the two. The following diagram shows those three components and their ports (i.e. which information they pass on):

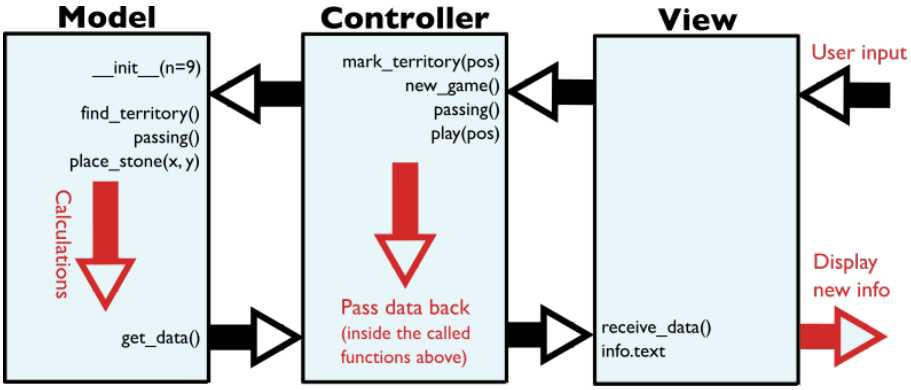

Figure 1: The View gets input from the user and passes it to the Controller by calling the corresponding methods. The Controller feeds the model, gets the results back and returns the new data to the View, which can then display the updated situation.

The parts Model and View have each a task 0. Those have to be done by both persons working on that part (these task comprise mainly copying over the base structure from the template). Afterwards the two persons can split up, one doing task 1 and the other task 2. At the end the need to merge their code together.

Nevertheless it might be very useful to communicate with each other an exchange the code regularly.

The person writing the controller is in the end responsible to import the View and the Model and to test the entire application.

Make sure to comment your code very detailed and to write docstrings for all methods, classes, and modules you implement. A docstring could look like:

```
"""One line summary of the function.
   Author: J.
   Further more detailed description if necessary

   Arguments:
       radius (float): radius of the standard sphere
       n (int): dimension

    Return:
       (float): volume of the sphere
"""
```

5.1 View

Task 0: Preparations

A first window Open the file `client.py`.
Explanation: It should contain one class which is a basic setup for a window in Pyglet. The three important methods for this class are:

(a) `__init__()` : gets called once when you initialise the window.

(b) `on_draw()` : gets called whenever the window updates its graphics.

(c) `on_mouse_press()` : gets called whenever the user clicks somewhere inside the window

(d) `on_key_press()` : gets called whenever the user presses any key on the keyboard.

(e) `update()` : can be called whenever the window has to update it's data.

Note: We separate the calculations (`update()`) and the drawing (`on_draw()`). In this way the program runs much smoother.

Execute the code and check that at least a window appears (maybe with strange content).

Shared data with the Controller In the `__init__()` method, create a dictionary called `self.data`. This dictionary will contain all gameplay informations that come from the controller:

```
self.data = {
            'size'     : n,  #n comes as keyword-argument to __init__()
            'stones'   : [[None for x in range(n)] for y in range(n)],
            'territory': [[None for x in range(n)] for y in range(n)],
            'color'    : None,
            'game_over': False,
            'score'    : [0, 0]}
```

Initialize the graphics Pyglet uses batches to gather all graphical elements together and draw them simutaneously. Create a method named `init_display()`.
Call it at the end of the method `__init__()`.
In `init_display()`, create a Pyglet batch, named `self.batch`, and draw that batch in `on_draw()`:

```
# Creating a batch [in init_display()]
self.batch = pyglet.graphics.Batch()
# Drawing the batch (which does not contain any graphics yet) [in on_draw()]
self.batch.draw()
```

Furthermore call `self.clear()` at the very top of the method `on_draw()` in order to clear out old graphics before it draws new ones.

If you run the program now, it should open a clean (black, white or gray) window.
(Hint: You can use `pyglet.gl.glClearColor(0.5,0.5,0.5,1)` in `self.__init__()` to change the default color)

Basic pyglet window:

```
import pyglet

# constants
BLACK = True
WHITE = False

class Window(pyglet.window.Window):

    def __init__(self):
        super(Window, self).__init__(700, 700, fullscreen=False, caption='')

    def on_draw(self):
        """Draw the interface."""
        pass

    def on_mouse_press(self, mousex, mousey, button, modifiers):
        """Function called on any mouse button press."""
        pass

    def on_key_press(self, symbol, modifiers):
        """Function that gets called on any key press (keyboard)."""
        pass

    def update(self, *args):
        """This function does all the calculations when the data gets updated.

        Side note: Has to be called manually.
                   For other games that require permanent simulations you would add
                   the following line of code at the end of  __init__():

                   pyglet.clock.schedule_interval(self.update, 1/30)
        """
        pass
if __name__ == '__main__':
    window = Window()
    pyglet.app.run()
```

Now you're ready for the real tasks. One person can do Task 1 and the other Task 2.

5.1.1 Task 1: The Board

Subtask 1: Drawing the background

- You can load images with `pyglet.resource.image()`.
  In the `__init__()` method, load the background image:

```
# Example:
self.image_background = pyglet.resource.image('images/Background.png')
```

- Sprites are used to display images in your window. Import `pyglet.sprite.Sprite` (at the top of the file). In `init_display()`, create a Sprite with the image of the background. You can specify the batch as keyword argument. In this way the sprite will the be drawn automatically when the entire batch is drawn.

```
#Import
from pyglet.sprite import Sprite
# Example usage
self.background = Sprite(self.image_background, batch=self.batch)
```

If you run the code now, you should see the image as background.

Additional Even though the Sprite is assigned to the batch (and will be drawn automatically as part of the batch), we need a active identifier to the Sprite object (otherwise Python would delete it).
Create an empty list `self.graphical_obj = []` in the method `init_display()`, where you can dump in all graphical objects that you don't need to change later. I.e. you can then do the following instead of defining `self.background` as you did above:

```
self.graphical_obj.append(Sprite(self.image_background, batch=self.batch))
```

Subtask 2: Grid layout and first user interaction In this section we will add the basic grid (On which the stones are placed) to the game and check if the user clicks on a intersection point.

- Pyglet uses ordered groups to define in which order elements in a batch are drawn (see example below).
  Add ordered groups `self.grp_back`, `self.grp_grid`, `self.grp_label`, `self.grp_stones`, and `self.grp_terrritory` (increasing order). Add the background sprite from before to the first of those groups.

```
# Example:
# Ordered groups are like different layers inside the batch. The lowest number
# will be drawn first.
# Inside a group the order is arbitrary (gives Pyglet the opportunity to optimize).
self.grp_back = pyglet.graphics.OrderedGroup(0)
self.grp_fore = pyglet.graphics.OrderedGroup(1)
# Just add the group as keyword argument to the graphical object:
self.background = Sprite(self.image_background, batch=self.batch, group=self.grp_back)
self.foreground = Sprite(image_foreground, batch=self.batch, group=self.grp_fore)
```

- Import the class `Grid` from the module `graphics.py`. You can then look at the docstrings (`help(Grid)`) to figure out which arguments it takes.
  Create a grid (named `self.grid`) in the method `init_display()`. Make sure to specify the batch and the group, so that it will be drawn correctly.

```
# In order to get the help you can either open the graphics.py and
# look at the docstrings or start python and use help():
from graphics import Grid
help(Grid)

# Note: use `self.data['size']` as the size (argument `n`) of the grid
# Hint: self.width and self.height store the size of the Window.
```

Now to the first user interaction. the method `on_mouse_press()` gets called whenever the user clicks inside the window. The follow example shows how to test for a left-click:

```
def on_mouse_press(self, mousex, mousey, button, modifier):
    if button == pyglet.window.mouse.LEFT:
        print('Left-click at position x={}, y={}'.format(mousex, mousey))
```

- Test for a left-click, get the field indices from `pos = self.grid.get_indices(mousex, mousey)` and print a notification if `pos` is not `None`.
- To wrap up, check in `self.update()` if `self.data['size']` is `self.grid.size`. If it is, call `self.init_display()` to recreate the grid of the new size.

Note: You can get more informations about different mouse buttons and keyboard buttons with `help(pyglet.window.mouse)` and `help(pyglet.window.key)`

Subtask 3: Drawing the stones The method `self.update()` gets called whenever something changes. Therefore, we need to create all graphical objects that can change over time inthere. graphical objects.

- In `self.update()`, create a new batch `self.batch_stones = pyglet.graphics.Batch()` and an empty list `self.stone_sprites`.
- Make sure you load the images for the stones (in the same way as you loaded the background image). You can center the images with the function below:

```
def center_image(image):
    """Sets an image's anchor point to its center"""
    image.anchor_x = image.width/2
    image.anchor_y = image.height/2

# Example usage:
center_image(self.image_black_stone)
# Hint: this all happens in the init_display()
```

- Iterate over all fields of `self.board` and create a stone sprite at each position where a stone is.
  Some remarks:

(a) `self.board[j][i]` is `None` if the field is empty

(b) Otherwise `self.board[j][i].color` is either `BLACK` or `WHITE` (booleans).

(c) `self.grid.get_coords(i, j)` will give you the coordinates in pixels.

(d) `self.grid.field_width` is the width between two lines of the grid. `self.image_black_stone.width` is the size of the image. Use both to scale the stones.

```
# Example: Create a stone sprite and scale it.
self.stone_sprites = []
_s = Sprite(..., batch=self.batch_stones, group=self.grp_stones)
_s.scale = 1./4
self.stone_sprites.append(_s)

# Tip: you can temporary change the values in self.data['stones'] to
# True or False to test it.
```

- Import `Circle` from `graphics.py`.
- Do the exact same thing again for the territory markers. Just check this time if the value `self.data['territory'][j][i]` is `BLACK`, `WHITE` or `None`. Then draw a `Circle()` instead of creating a sprite. Make sure the circles are drawn above the stones.

```
if self.data['territory'][j][i] == Black:
    Circle(x, y, color=(0,255,0,255), r=100 batch=self.batch_stones,
           group=self.grp_territory)
```

- Finally, display one stone in the very topright corner to show whose turn it is (you get this info from `self.data['color']`).
  You'll need to speak with your partner, who had created the Labels, to position this stone correctly.

5.1.2 Task 2: Buttons and Labels

Subtask 1: communication with the controller The View gets all its informations and data from the Controller.

- Create a method `receive_data(self, data)` that takes a dictionary and updates the dictionary `self.data` with the new one and then calls `self.update()` (updating a dictionary will update all keywords that are present in the new one without deleting missing ones).

```
self.data.update(data)
self.update()
```

- Now create a new class called `DummyController` which has prototypes of all the functions that the controller would have. At the moment all those functions can just print something (like `'Played at position x, y.'`).

```
class DummyController(object):
    def mark_territory(self, pos):
        # code
    def new_game(self):
        # code
    def passing(self):
        # code
    def play(self, pos):
        # code
```

- Add a keyword-argument `controller` to the `__init__()` of the class `Window`. Then in `__init__()`, create an attribute `self.controller` that takes either the value from the keyword-argument or a new instance of `DummyController` if the keyword-argument is not specified.
  In this way you can test your code even if your teammate who writes the Controller has not yet finished.

Subtask 2: Displaying informations Additional to the main game elements we need some informations displayed at the border of the bord. Text can be displayed by using Labels:

```
from pyglet.text import Label

# Example usage
Label(x=10, y=10, text='Score:', color=(0, 0, 0, 255),
      font_size=12, batch=self.batch, group=self.grp_label)
```

```
class DummyController(object):
    def mark_territory(self, pos):
        # code
    def new_game(self):
        # code
    def passing(self):
        # code
    def play(self, pos):
        # code
```

- Create 7 labels. Five to display the score, one for showing who's turn it is, and one showing informations from the Controller. See the illustation below for an example. The one showing the informations must be named `self.info`, the two labels displaying the score should be called `self.score_black` and `self.score_white`.

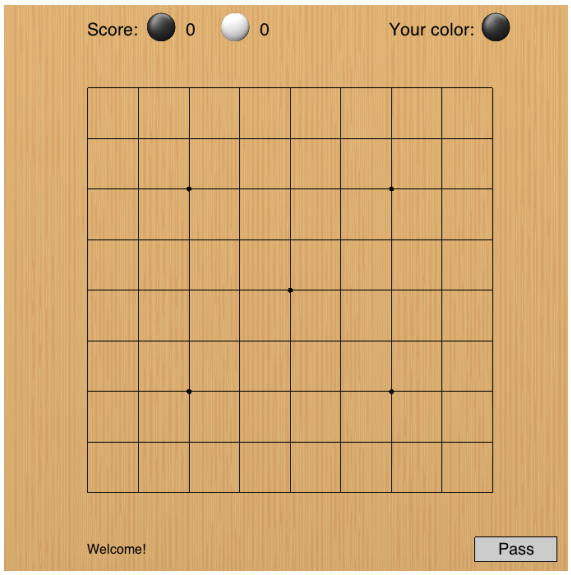

Figure 2: Example of how the board could look like

Note: For the score display just create two additional Labels saying '(Black)' and '(White)' instead of the two stones you see in the picture.

Subtask 3: Buttons Now we need some buttons that can be pressed. Use the already implemented `Button()` from the `graphics.py` for that.

- Import `Button` from `graphics.py`. Add a `Button` (in `init_display()`) saying `'Pass'` (name it `self.button_pass`).
- In `on_mouse_press()`, check for clicks on that button and call `self.controller.passing()` in such a case.

```
from graphics import Button
b = Button(pos=(10,10), text='Button', batch=self.batch)

# And then in on_mouse_press():
if (mousex, mousey) in b:
    print('The button has been pressed')
```

- Do the same for a second Button saying 'New game'. However, do not add this one to `self.batch`! Instead, check in `on_draw()` if `self.data['game_over']` is `True` and if it is, call `self.button_newgame.draw()` explicitly.
- Check also for clicks on that button (but only if the game is over).
  This should call `self.controller.new_game()`.

Subtask 4: Updating labels We need to update the texts of the labels. You can do that by simple set a new value to their attribute `.text`:

```
self.score_black.text = '12'
```

- In the method `self.update()`, update the labels `self.score_black` and `self.score_white`. You find the values in `self.data['score']`.

Finishing touches (both together)

Merge your parts together. Maybe you need to adjust the position of some graphical elements slightly.
Task 1 has checked in `on_mouse_press()` for a click in the grid. There you need to call either `self.controller.play(pos)` or `self.controller.mark_territory(pos)`, depending if the game is over or not.

Make sure you indicate (comments) which part of the code comes from which person.

Additional:

Replace the two labels saying '(Black)' and '(White)' in the score display with two sprites displaying the corresponding stones.

5.2 Controller

- Create a class `Controller()`. At the bottom of the file, add a `if __name__ == '__main__':` part, where you create one instance of `Controller`.
- Create two dummy classes `DummyView` and `DummyModel` in order to test your program before the other team members are done with their part.

```
class DummyWindow(object):
    def receive_data(self, data):
        pass
    class Label:
        self.text = ''
    self.info = Label()

class DummyModel(object):
    def __init__(n=19):
        self.data = {'size'      : n,
                     'stones'    : [[None for i in range(n)]for j in range(n)],
                     'territory': [[None for i in range(n)]for j in range(n)],
                     'game_over': False,
                     'score'     : (0,0),
                     'color'     : True}
    def find_territory(self):
        pass
    def get_data(self):
        return self.data
    def mark_territory(x, y):
        pass
    def passing(self):
        return False
    def place_stone(self, x, y):
        return False
```

- Create the `__init__()` method of your class `Controller`.
  it should first create an instance of `Window` (`DummyWindow` at the moment) and one of `Model` (rsp. `DummyModel`). Call them `self.window` and `self.model`. Note that `Model` takes an argument `n=9` that determines the board size.

Then the `self.__init__()` should get the data from the model (with `self.model.get_data()`) and pass it to the window (with `self.window.receive_data()`).
Finally call `pyglet.app.run()` This would start the Pyglet window.
(You'll need `import pyglet` at the very top)

- Create a function `self._update_window()` that pass on the data from the model to the window.
- Create three methods `self.play(pos)`, `self.passing()`, and `self.mark_territory(pos)`. Those can be called by the user via the interface.
- `self.play(pos)` should call the `play` function of the model. If it returns `True`, you should display an info saying whose turn it is. If it return `False`, you should display a message 'invalid move!'

```
# You can display a message just by setting the following string:
self.window.info.text = 'important message'
```

- `self.passing()` should also call the corresponding method of self.model. If it succeeded (returns True), it should check if the game is over. In either case it should print the correct information to self.window.info.text

```
# You can get if the game is over from
self.model.get_data()['game_over']
```

- `self.mark_territory(pos)` also just calls the corresponding function from the model.

Make sure all those functions call at the end `self._update_window()`

- If the other groups finished their part, you can import them and replace your Dummy-classes with the actual ones. Test the game by starting your controller.py file.

  ```
  # E.g.
  from game_model import Model
  ```

Help on starting the game:

```
# Go into the proper directory
cd /Users/you/path/to/your/directory
# Execute your code using Python 2
python ./controller.py # or `python2` or `python2.7`
```

Make sure all .py files are in the same folder, and also that the images/ folder is in there as well.

5.3 Model

Task 0: Get familiar with the rules of Go.

In Go the players place stones on the board in alternating turns. Stones of the same color that connect to each other form a group.

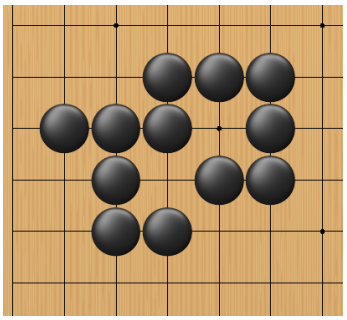

Figure 3: A group consists of directly connected stones of one color.

A group dies (or gets captured) if it is completely surrounded by enemy stones (or the game border). In the picture below, white can kill the black group by placing a stone on the middle field. A dead group gets removed from the board and each stone counts one point.

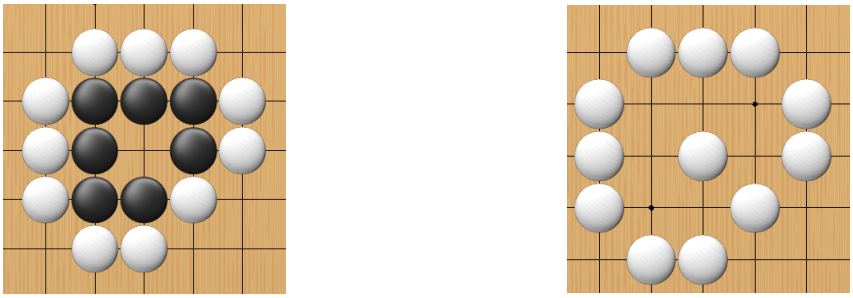

Figure 4: White can kill the black group by placing a stone on the last free field adjacent to the black group.

Therefore you can easily see that a group is save, if it has at least two so called eyes (an eye is one connected area that is completely surrounded by one color. An eye can contain multiple fields.) The group below has two eyes and therefore can't be killed anymore (since the opponent can not place on both places at the same time).

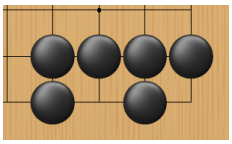

Figure 5: White cannot place in both eyes simultaneously. Therefore the black group is save.

The goal of Go is to place your stones in such a way that you claim as many empty fields for your side as possible. Each empty field counts one point.
Empty fields are counted as points for your side, if the enemy cannot (or wont) place stones inside that area because such placed stones would be killed again.
Stones that are dead will be removed at the end of the game, giving one point for the captured stone and another one for the empty field.

In the picture below you see how a final board could look like. Both sides have some stones that are clearly dead. The points on the empty fields indicate whose territory it is (i.e. one point per marker on an empty area and two points for a marker on an enemies stone). Additionaly, the stones that have been captured during the game count 1 point each as well.

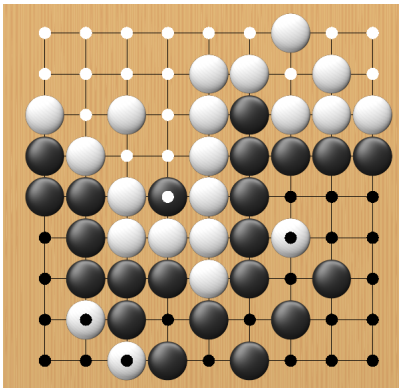

Figure 6: The dots mark the points. An empty field gives 1 point, each captured stone gives 2 (1 for the stone and then 1 for the field). Then the stones captured during the game also count 1 point each.

- Open the game_model.py.
- At the top of the file, import `Group` from `template.py` and define two constants `BLACK=False`, `WHITE=True`.

```
from template import Group

BLACK = False
WHITE = True
```

- Create a class called `Model`.

Now you're ready to roll. One person can do Task 1, the other Task 2. If you're done there are two additional tasks available. Additional 1 is not absolutely necessairy, but it's a very short task anyway. Additional 2 is the entire part to count the score at the end. There is an instruction on how to import that piece from the template (That means the solutions are already in `template.py` but you can try it yourself).

Task 1: Helper methods, passing method

- In the `__init__()` method, add all neaded attributes. It takes one keyword-argument `n=11`.

Important is to understand the structure of `self.board`. It is a "(n x n)-matrix" each field contains either `None` or a `Group`. Since the `Groups` are mutable objects, we can make that each field where the group is (where a stone of the stone group lies) points to the same `Group` object. Furthermore, this has the nice property that if we remove all links in `self.board`, Python will delete that group automatically.

```
# Gameplay attributes
self.size = None            # should be an integer, passed to the __init__()
                            # as keyword argument (default: 11)
```

```
self.turn = BLACK           # boolean
self.blocked_field = None   # Used for the Ko rule
self.has_passed = False     # To detect if both players pass.
self.game_over = False

# The board is represented by a SxS-matrix where each entry
# contains the information which stone lies there.
self.board = [[None for i in range(self.size)] for j in range(self.size)]
self.territory = [[None for i in range(self.size)] for j in range(self.size)]

# We need to count the captured stones that got removed
# from the board
self.score = [0, 0]      # score from empty fields at the end of the game.
self.captured = [0, 0]   # stones killed during the game
```

- *(Pass function)* Write a function `self.passing()` that does the following:
  If `self.has_passed` is `False`, it sets `self.turn` to the other color, `self.has_passed` to `True` and `self.blocked_field` to `None`.
  If `self.has_passed` is `True`, it sets `self.game_over` to `True`. In both those cases the function should return `True`. Furthermore, add at the top of that function a check if `self.game_over` is `True`. If it is, the function should return `False` (without doing anything).
- *(Getting stone colors)* Write a helper function `self._stones()` that returns a nested list (same shape as the `self.board`) that contains only the colors of the stones. In particular, the function should loop over the `self.board`. If `self.board[j][i] is None`, it should set `stones[j][i]` to `None`, else it should set `stones[j][i]` to `self.board[j][i].color`.
- *(prepare data for the GUI)* Write a function `self.get_data` that returns a dictionary containing all important informations for the GUI. See example below:

```
data = {'size'       : self.size,
        'stones'     : None,           # Use here the result from your function
                                       # self._stones()
        'territory'  : self.territory,
        'game_over'  : self.game_over,
        'score'      : None,           # to be implemented: The sum of self.score
                                       # and self.captured
        'color'      : self.turn}
```

- create a method `self._add(grp)` which takes a `Group` and for each coordinate tuple in `grp.stones` it should add the group `grp` to `self.board`:

  ```
  self.board[y][x] = grp  # for each elem. (x,y) in grp.stones
  ```

- create another method `self._remove(grp)` that does the exact opposite. For each `(x,y)` in `grp.stones` it should set the corresponding field in `self.board` to `None`.
- Create a third method `self._kill(grp)` that first increases the opponent's score in `self.captured` by the number of stones in the group and then removes that group (with the method implemented before).
- As a last task, create a method `self._liberties(grp)` that returns the number of free places next to a group. For that you can iterate over all `(x,y)` in `grp.border` and check if the `self.board` is `None`.

Task 2: Place stone

We receive the coordinates `(x,y)` where a stone should be placed. We store adjacent stones together as one `Group`, in this way it's easier to find out if a stone should be removed and we can simply remove the entire group. The group class is already implemented. You can import it with `from template import Group`.

A group has a attribute `stones`, that contains coordinate tuples of all stones and an attribute `border`, which contains the coordinates of all adjacent fields. In this way we can just iterate over the `border` to find out if a group still has a free space adjacent to it. Furthermore the group has a method `__add__()` that defines how we add two groups.

So our plan here is to create a new Group containing only the newly placed stone, then we iterate over all adjacent fields to find out if the move is valid. If it is we actually place the created group on the board. For each adjacent field there are 4 possible cases. if it is empty we know that the move is valid. If it is part of a group of the same color, we merge this group into our new group and store the old group to be removed. If there is a group of the opponents color we can either kill it (if this is the last free adjacent place for it; this will make the move valid) or we can just leave it. If we still don't know whether the move is valid or not, we finally check if the newly created group has free adjacent spaces (checking which fields stored in `grp.border` are empty).

- Import `Group` from `template.py`.
- Create a method called `self.place_stone(x, y)`.
  Check if the game is over and return `False` in that case. Next, check if the field is free (i.e. if the corresponding field in `self.board` is `None`). Again, if it is not, return `False`.
- Now create a new stone group containing only the newly placed stone, and create a two lists `groups_to_remove` and `groups_to_kill` to save which groups need to be removed/killed (we need to wait with removing/killing them until we actually know the move is valid).

  ```
  new = Group(stones=[(x, y)], color=self.turn)
  groups_to_remove = list()
  groups_to_kill = []
  ```

  Iterate over all direct neighbors of the field `(x,y)`. Before you start the loop, set up a variable `is_valid = False`. We will set this to True as soon as we're sure the move is valid.
- First check if the neighbor is actually on the board (if not use `continue`).
- Add the neighbor to the border of our new group (we need that in order to calculate if a group is dead):

  ```
  new.border.add((u, v))  # Sets are similar to lists, but they only contain
                          # unique elements.
                          # E.g. set([2,2,1]) would be a set with only two
                          # elements: set([2,1])
                          # you can use set.add(...) to add an element
                          # (Would be list.append(...) for lists)
  ```

- Get the entry from `self.board` at the neighbors position. If it is `None`, set `is_valid` to `True`. Otherwise check if the both groups have the same color. In that case, you can merge the other group into our new group and add the other group to the groups we have to remove at the end.
  (Note: We're not doing anything on the board yet, we're just preparing to execute the move if it is valid. But if it is not, we can simply delete our new group and empty the list/set (`groups_to_remove`) and everything will be fine).

```
if other.color == new.color:
    new = new + other
    groups_to_remove.append(other)
```

- Otherwise the groups have different colors. Check if the liberties of the other group is 1 (i.e. The group has only one empty field next to it left). If that's the case, set `is_valid` to `True` and add the group to the `groups_to_kill`. Make sure `groups_to_kill` does not already contain the group you want to kill!

```
# The method self.liberties(grp) returns the number of free fields adjacent to
# the group. This method has been implemented by the team member doing subtask 1
elif self._liberties(other) == 1:
    is_valid = True
    # add the group to `groups_to_kill`
```

- Outside of that loop. We check first if the new group has at least one liberty `self._liberties(new)`. In that case set `is_valid` to `True`.
- Now check if the move is actually valid. If it is, remove all groups in `groups_to_remove` and kill all the groups in `groups_to_kill`. Then add the new group.

```
# You can just use those methods (defined the person doing subtask 1):
self._remove(grp)
self._kill(grp)
self._add(grp)
```

- To finish up, set `self.has_passed` to `False`, change `self.turn` to the other one and return True.

Additional 1: Ko Rule

In Go it's not allowed to immediately kill another single stone that has just catched a single stone (Because this would potentially end in an infinite loop). In particular, in the situation below, if white captures in picture 2 the single black stone, then black cannot capture the placed white stone immediately.

Figure 7: White can capture the black stone but black is not allowed to strike back immediately (because this would end in the same situation we started from)

First check in `place_stone()` if `self.blocked_field` is the give coords `(x,y)`. If it is, return `False`.

Now it remains to block the field after a stone has been placed. So at the bottom of the `self.place_stone()` method add a block of code that sets the `self.blocked_field` to some `(x,y)` if the Ko-rule applies.
In particular test if all of the following conditions are fullfilled: - the new stone group (`new`) has only one stone - Only one group will be killed (is inside `groups_to_kill`) - This one group has

only one stone If all are satisfied, set `self.blocked_field` to the proper coordinates (of the killed stone), else set it back to `None`.

```
# Tip: For a group grp.stones is a set.
#      In order to get elements from a set you have to iterate over it.
#      (since a set is not ordered and therefore set[0] would not make any sense)

#      this can be done with
for a in my_set:
    # ...

# or using a generator together with the next() command:
next(iter(my_set))
```

Additional 2: Mark territory

This task is optional. If you ran out time you can just add the following import on top of your file and make your `Model`-class inherit from the `Terr_Template`-class.
In this way, all methods in `Terr_Template` are accessible from your model as well.

```
from template import Terr_Template

class Model(Terr_Template):
    # ...
```

Here is what needs to be done, if you decide to tackle this task. All that's left for a complete game is to actually count the score when the game is over. So we need a function that the user can select which territory should be counted for whom.
Note: you can also just implement some of the 6 functions below and import the others using the trick above.

- Create a method `self._compute_score()` that iterates over `self.territory` and counts the number of marked fields.

In Particular, it should update `self.score`, adding 1 to `self.score[BLACK]` for each field where `self.territory[j][i] == BLACK` and similarly for white. Also, it should add 2 point (instead of 1) if the field is marked and `self.bord[j][i]` is not `None`.

- Create a recursive function `self._claim_empty(x, y, color)` that marks the given field (if it is empty) with the pecified color. Check recursively all neighboring fields and mark them too, if there empty.

```
self.territory[y][x] = color
# Tip: You'll need to keep track of all the fields you have visited.
#      for that add another keyword-argument `area = None`, which you then
#      initialize with an empty list. In this list you can put all coordinates where
#      you already where.
```

- Create a recursive function `self._claim_group(x, y, color)`. First it should claim the entire group (set territory value for each stone in `grp.stone`). Then it should call `self._claim_empty()` on each field in `grp.border` that is `None`.

- Now create a `_find_empty()` method. This is essentially the same as `self._claim_empty()` but it does not actually mark (claim) the territory but instead return the following things:
  - count (list [0,0]) where you count the adjacent stones of each color
  - area (list) List of all connected empty fields

Note: This function will be used to automatically mark territory that is surrounded by only one color.

- Furthermore, we need a function which which the player can mark territory.
  Create `self.mark_territory(x, y)`. If the game is not over it should not do anything.
  If the field `(x,y)` is empty it should cycle through the colors `[BLACK, WHITE, None]` (depending of the fields current territory marking) and then call `self._claim_empty(x,y,color)`.
  If the field has a group on it, it should either mark or unmark it calling `self._claim_group(x,y,color)`.
  At the end it should call `self._compute_score()`.
- Finally, create one more method `self.find_territory()`. It should iterate over the entire board (keeping track of where it already has been) and for each empty field it should call `self._find_empty()`.
  If `self._find_empty()` returns a `count` that says that the empty area is completely surrounded by one color, then call `self._claim_empty()` on that area.
  At the end call `self._compute_score()`