pygame.init()
pygame.init() initialize all imported pygame modules is a convenient way to get everything started

pygame.font.init
initialize the font module

since we can initialize all modules by the above call, I don't think font.init() is necessary

pygame.font.SysFont("comicsans", size, bold=True)

create a Font object from the system fonts

```
SysFont(name, size, bold=False, italic=False) -> Font
```

Return a new Font object that is loaded from the system fonts. The font will match the requested bold and italic flags. If a suitable system font is not found this will fall back on loading the default pygame font. The font name can be a comma separated list of font names to look for.

font.render(text, 1, color)
render()
draw text on a new Surface
render(text, antialias, color, background=None) -> Surface
This creates a new Surface with the specified text rendered on it. pygame provides no way to directly draw text on an existing Surface: instead you must use Font.render() to create an image (Surface) of the text, then blit this image onto another Surface.

The text can only be a single line: newline characters are not rendered. Null characters ('x00') raise a TypeError. Both Unicode and char (byte) strings are accepted. For Unicode strings only UCS-2 characters ('u0001' to 'uFFFF') are recognized. Anything greater raises a UnicodeError. For char strings a LATIN1 encoding is assumed. The antialias argument is a boolean: if true the characters will have smooth edges. The color argument is the color of the text [e.g.: (0,0,255) for blue]. The optional background argument is a color to use for the text background. If no background is passed the area outside the text will be transparent.

The Surface returned will be of the dimensions required to hold the text. (the same as those returned by Font.size()). If an empty string is passed for the text, a blank surface will be returned that is zero pixel wide and the height of the font.

Depending on the type of background and antialiasing used, this returns different types of Surfaces. For performance reasons, it is good to know what type of image will be used. If antialiasing is not used, the return image will always be an 8-bit image with a two-color palette. If the background is transparent a colorkey will be set. Antialiased images are rendered to 24-bit RGB images. If the background is transparent a pixel alpha will be included.

Optimization: if you know that the final destination for the text (on the screen) will always have a solid background, and the text is antialiased, you can improve performance by specifying the background color. This will cause the resulting image to maintain transparency information by colorkey rather than (much less efficient) alpha values.

If you render '\n' an unknown char will be rendered. Usually a rectangle. Instead you need to handle new lines yourself.

Font rendering is not thread safe: only a single thread can render text at any time.

surface.blit(label, (top_left_x + play_width /2 - (label.get_width()/2), top_left_y + play_height/2 - label.get_height()/2))

Putting this in real terms may help, although as simply put as possible -> blitting is drawing

Going through each of the steps you have mentioned:

Create a canvas of a desired size
This is our window, created by screen = pygame.display.set_mode((width,height)). Where screen is the canvas name. Eventually everything will need to be drawn onto this canvas so that we can see it.

Create a surface of smaller size containing the object to be displayed
This is a surface that we will populate with objects such as images. It does not need to be smaller than the window size and it can be moved around freely.

Define a Rect value of the surface
When you create a surface using something like background = pygame.Surface((width,height)) you specify it's size. The images or drawn items on the surface can be any shape or size but they must all be contained within the bounds set by this width and height.

Blit (overlap) the surface on the canvas at the rect position
Now the all important bit. We need to get this surface (background) and draw it onto the window. To do this we will call screen.blit(background,(x,y)) where (x,y) is the position inside the window where we want the top left of the surface to be. This function says take the background surface and draw it onto the screen and position it at (x,y).

pygame.draw.line(surface, GREY_COLOR, (sx, sy + i*block_size), (sx+play_width, sy+ i*block_size))

`pygame.draw.line()`

draw a straight line

```
line(surface, color, start_pos, end_pos, width) -> Rect
line(surface, color, start_pos, end_pos, width=1) -> Rect
```

Draws a straight line on the given surface. There are no endcaps. For thick lines the ends are squared off.

Parameters:
- **surface** (***Surface***) -- surface to draw on
- **color** (***Color*** *or int or tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple `(RGB[A])`
- **start_pos** (*tuple(int or float, int or float) or list(int or float, int or float) or* ***Vector2****(int or float, int or float)*) -- start position of the line, (x, y)
- **end_pos** (*tuple(int or float, int or float) or list(int or float, int or float) or* ***Vector2****(int or float, int or float)*) -- end position of the line, (x, y)
- **width** (*int*) --

(optional) used for line thickness

if width >= 1, used for line thickness (default is 1)
if width < 1, nothing will be drawn

Note: When using `width` values `> 1`, lines will grow as follows.

For odd `width` values, the thickness of each line grows with the original line being in the center.

For even `width` values, the thickness of each line grows with the original line being offset from the center (as there is no exact center line drawn). As a result, lines with a slope < 1 (horizontal-ish) will have 1 more pixel of thickness below the original line (in the y direction). Lines with a slope >= 1 (vertical-ish) will have 1 more pixel of thickness to the right of the original line (in the x direction).

Returns: a rect bounding the changed pixels, if nothing is drawn the bounding rect's position will be the `start_pos` parameter value (float values will be truncated) and its width and height will be 0

pygame.draw.rect(surface, (255, 0, 0), (top_left_x, top_left_y, play_width, play_height), 5)

draw a rectangle

```
rect(surface, color, rect) -> Rect
rect(surface, color, rect, width=0, border_radius=0,
border_radius=-1, border_top_left_radius=-1,
border_top_right_radius=-1, border_bottom_left_radius=-1) -> Rect
```

Draws a rectangle on the given surface.

Parameters:
- **surface** (***Surface***) -- surface to draw on
- **color** (***Color*** *or int or tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple `(RGB[A])`
- **rect** (***Rect***) -- rectangle to draw, position and dimensions
- **width** (*int*) --

  (optional) used for line thickness or to indicate that the rectangle is to be filled (not to be confused with the width value of the `rect` parameter)

  if `width == 0`, (default) fill the rectangle
  if `width > 0`, used for line thickness
  if `width < 0`, nothing will be drawn

  Note: When using `width` values `> 1`, the edge lines will grow outside the original boundary of the rect. For more details on how the thickness for edge lines grow, refer to the `width` notes of the `pygame.draw.line()` function.

- **border_radius** (*int*) -- (optional) used for drawing rectangle with rounded corners. The supported range is [0, min(height, width) / 2], with 0 representing a rectangle without rounded corners.
- **border_top_left_radius** (*int*) -- (optional) used for setting the value of top left border. If you don't set this value, it will use the border_radius value.
- **border_top_right_radius** (*int*) -- (optional) used for setting the value of top right border. If you don't set this value, it will use the border_radius value.
- **border_bottom_left_radius** (*int*) -- (optional) used for setting the value of bottom left border. If you don't set this value, it will use the border_radius value.

value, it will use the border_radius value.

- **border_bottom_right_radius** (*int*) --

(optional) used for setting the value of bottom right border. If you don't set this value, it will use the border_radius value.

if `border_radius` < `1` it will draw rectangle without rounded corners
if any of border radii has the value < `0` it will use value of the border_radius
If sum of radii on the same side of the rectangle is greater than the rect size the radii
will get scaled

Returns: a rect bounding the changed pixels, if nothing is drawn the bounding rect's position will be the position of the given `rect` parameter and its width and height will be 0

Return type:Rect

pygame.time.Clock()
Creates a new Clock object that can be used to track an amount of time. The clock also provides several functions to help control a game's framerate.
create an object to help track time

clock.get_rawtime()

pygame.time.Clock.get_rawtime
actual time used in the previous tick

pygame.event.get()

```
pygame.event.get()
```

get events from the queue

```
get(eventtype=None) -> Eventlist
get(eventtype=None, pump=True) -> Eventlist
```

This will get all the messages and remove them from the queue. If a type or sequence of types is given only those messages will be removed from the queue.

If you are only taking specific events from the queue, be aware that the queue could eventually fill up with the events you are not interested.

If `pump` is `True` (the default), then `pygame.event.pump()` will be called.

pygame.display.quit()

`pygame.display.quit()`

Uninitialize the display module

`quit() -> None`

This will shut down the entire display module. This means any active displays will be closed. This will also be handled automatically when the program exits.

It is harmless to call this more than once, repeated calls have no effect.

pygame.display.update()

`pygame.display.update()`

Update portions of the screen for software displays

`update(rectangle=None) -> None`

`update(rectangle_list) -> None`

This function is like an optimized version of `pygame.display.flip()` for software displays. It allows only a portion of the screen to updated, instead of the entire area. If no argument is passed it updates the entire Surface area like `pygame.display.flip()`.

You can pass the function a single rectangle, or a sequence of rectangles. It is more efficient to pass many rectangles at once than to call update multiple times with single or a partial list of rectangles. If passing a sequence of rectangles it is safe to include None values in the list, which will be skipped.

This call cannot be used on `pygame.OPENGL` displays and will generate an exception.

win.fill((255,0,100))

pygame.display.set_mode((s_width, s_height))

Initialize a window or screen for display

pygame.display.set_caption('Tetris')

Get the current window caption

`get_caption() -> (title, icontitle)`

Returns the title and icontitle for the display Surface. These will often be the same value.