pythonGraph Overview

pythonGraph allows programmers to output graphic images to a window. In addition, it provides functionality to interact with the user via the mouse and via individual keystrokes on the keyboard.

You can download pythonGraph using the automated Python Package Management, pip. Alternatively, you can obtain the full source code at https://github.com/USAFA-CompSci110/pythonGraph.

Getting Started

To get started, import the pythonGraph library in your python file, as shown below:

```python
import pythonGraph
```

Then, open a pythonGraph window by calling the `open_window` function and specifying the dimensions of the window. A successfully opened window will appear with a white background.

pythonGraph utilizes a coordinate system where **<u>the origin (0, 0) is at the top-left hand corner</u>**. When the program requests the mouse's position or the location of a click, these will be given using the same coordinate system.

The picture below shows the coordinate layout for a pythonGraph window opened with `open_window(400,300)`.

Example Code

```python
# Opens a Blank Graph Window (400 x 300 pixels)
pythonGraph.open_window(400, 300)
```

Output

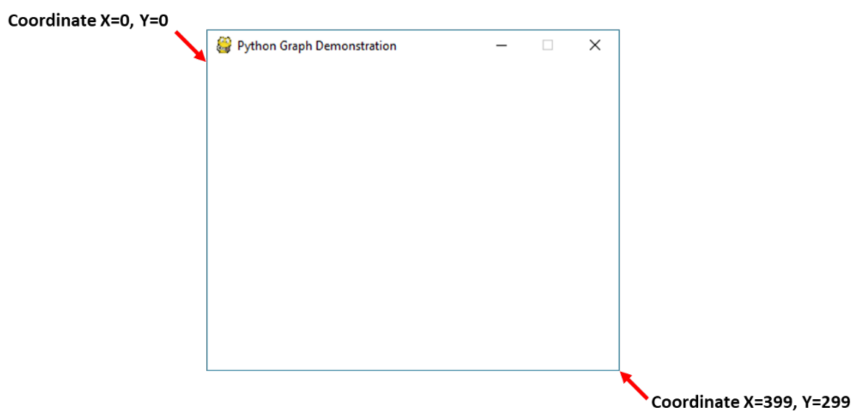


Table of Contents

1. Drawing Operations

pythonGraph's drawing routines can output a variety of shapes in a variety of colors.

Before using these operations, please note that:

- `open_window` must be called first, otherwise a run-time error will occur.
- You must call `update_window` before the result of the drawing routines will be visible on the screen.

METHODS DESCRIBED IN THIS CHAPTER

- clear_window
- draw_arc
- draw_image
- draw_rectangle
- draw_circle
- draw_ellipse
- draw_line
- draw_pixel
- draw_text

1.1 clear_window

Usage

```
clear_window(color)
```

Description

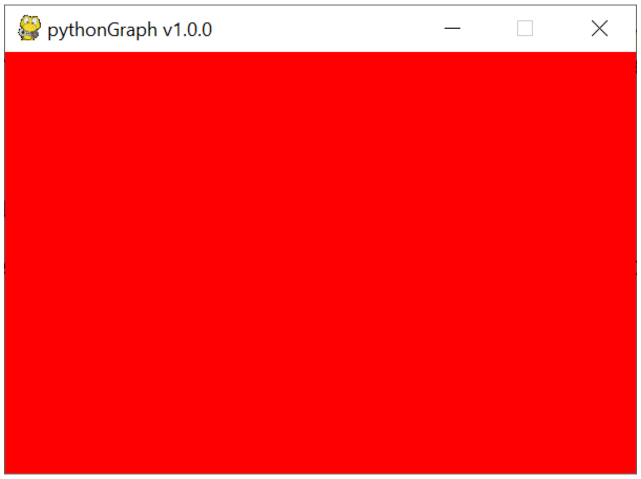
Clears the entire window to a particular color.

`color` can either be a predefined value (refer to `pythonGraph.colors`) or a custom color created using the `create_color` or `create_random_color` functions.

Example

```
pythonGraph.clear_window(pythonGraph.colors.RED)
```

Output



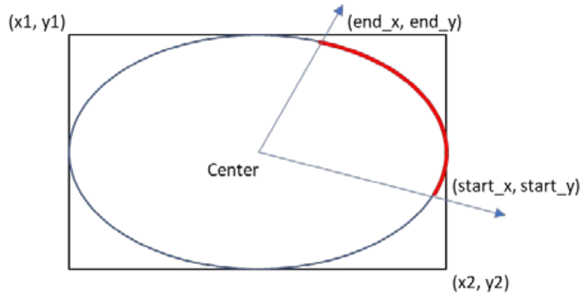
1.2 draw_arc

Usage

```
draw_arc(x1, y1, x2, y2, start_x, start_y, end_x, end_y, color, width)
```

Description

Draws a portion of the ellipse that is inscribed inside the given rectangle:



The parameters **`(x1, y1)`** and **`(x2, y2)`** represent the two opposite corners of the rectangle.

The arc begins at the intersection of the ellipse and the line passing through the center of the ellipse and **`(start_x, start_y)`**. It then proceeds counter-clockwise until it reaches the intersection of the ellipse and the line passing through the center of the ellipse to **`(end_x, end_y)`**.

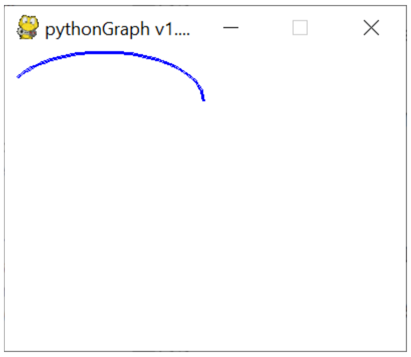
`color` can either be a predefined value (refer to **`pythonGraph.colors`**) or a custom color created using the **`create_color`** or **`create_random_color`** functions.

`width` is an optional parameter that specifies the "thickness" of the arc in pixels. Otherwise, it uses a default value of 2.

Example

```
pythonGraph.open_graph_window(400, 300)
pythonGraph.draw_arc(1, 100, 200, 1, 250, 50, 2, 2, pythonGraph.colors.BLUE, 3)
```

Output



1.3 draw_image

Usage

```
draw_image(filename, x, y, width, height)
```

Description

Draws an image in the pythonGraph window.

`filename` refers to the name of the file (*e.g.,* "image.png") to be drawn. You can use any BMP, JPEG, or PNG file. **The image file should be in the same folder as your python script.**

`x` and **`y`** specify the upper-left coordinate where the image is to be drawn.

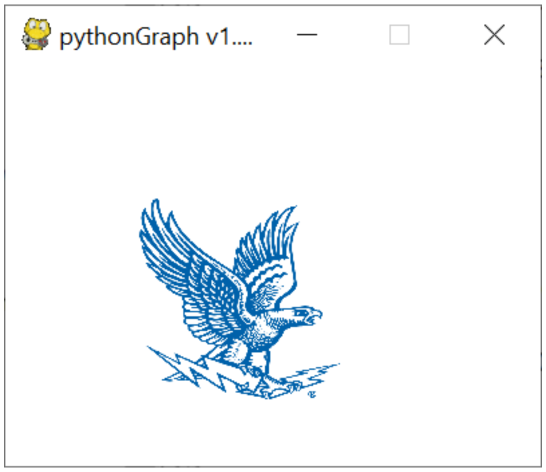
`width` and **`height`** represent the desired dimensions of the image. pythonGraph will try to scale the image to fit within these dimensions.

Example

For this example, assume that the file "falcon.png" exists.

```
pythonGraph.open_graph_window(400, 300)
pythonGraph.draw_image("falcon.png", 100, 100, 150, 150)
```

Output



1.4 draw_rectangle

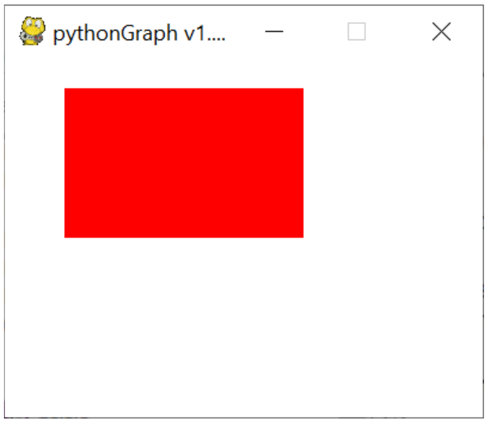
Usage
`draw_rectangle(x1, y1, x2, y2, color, filled, width)`

Description
Draws a rectangle on the screen. **`(x1, x2)`** is any corner of the rectangle **`(x2, y2)`** is the opposite corner of the rectangle **`color`** specifies the rectangle's color. This can either be a predefined value (refer to pythonGraph.colors) or a custom color created using the **`create_color`** function. **`filled`** can be either **`True`** or **`False`**, depending on whether or not the rectangle should be filled in or not, respectively. **`width`** is an optional parameter that specifies the width of the rectangle's border. If this value is not provided, a default value will be used.

Example

```
pythonGraph.open_window(400, 300)
pythonGraph.draw_rectangle(50, 150, 250, 25, pythonGraph.colors.RED, True)
```

Output



1.5 draw_circle

Usage

```
draw_circle(x, y, radius, color, filled, width)
```

Description

Draws a circle at **`(x, y)`** with the specified **`radius`**

`color` specifies the circle's color. This can either be a predefined value (refer to pythonGraph.colors) or a custom color created using the **`create_color`** function.

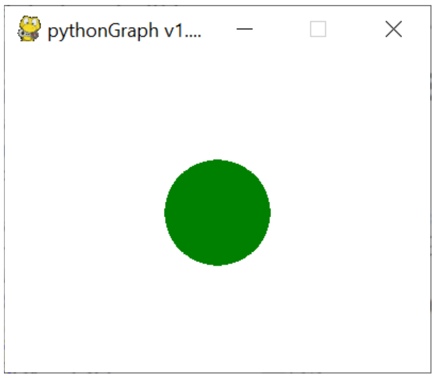
`filled` can be either **`True`** or **`False`**, depending on whether or not the circle should be filled in or not, respectively.

`width` is an optional parameter that specifies the width of the circle's border. If this value is not provided, a default value of 2 will be used. This parameter will be ignored if **`filled`** is **`True.`**

Example

```
pythonGraph.open_window(400, 300)
pythonGraph.draw_circle(200, 150, 50, pythonGraph.colors.GREEN, True)
```

Output

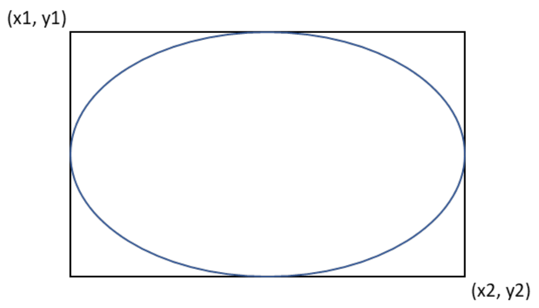


1.6 draw_ellipse

Usage
`draw_ellipse(x1, y1, x2, y2, color, filled, width)`

Description

Draws an ellipse inscribed in the rectangle whose two diagonally opposite corners, **`(x1, y1), (x2, y2)`** are given:



`color` can either be a predefined value (refer to **`pythonGraph.colors`**) or a custom color created using the **`create_color`** or **`create_random_color`** functions.

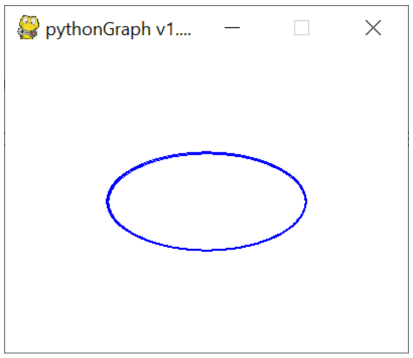
`filled` can be **`True`** or **`False`**, depending on whether or not the ellipse is filled in or not, respectively.

`width` is an optional parameter that specifies the width of the ellipse's border. If this value is not provided, a default value of 2 will be used.

Example

```
pythonGraph.open_window(400, 300)
pythonGraph.draw_ellipse(100, 100, 300, 200, pythonGraph.colors.BLUE, False, 4)
```

Output



1.7 draw_line

Usage

```
draw_line(x1, y1, x2, y2, color, width)
```

Description

Draws a line segment from **`(x1, y1)`** to **`(x2, y2)`** in the given color:

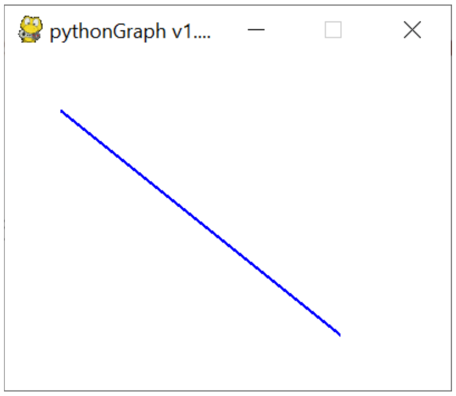
`color` can either be a predefined value (refer to **`pythonGraph.colors`**) or a custom color created using the **`create_color`** or **`create_random_color`** functions.

`width` is an optional parameter that specifies the width of the line. If this value is not provided, a default value of 2 will be used.

Example

```python
pythonGraph.open_window(400, 300)
pythonGraph.draw_line(50, 50, 300, 250, pythonGraph.colors.BLUE, 3)
```

Output



1.8 draw_pixel

Usage

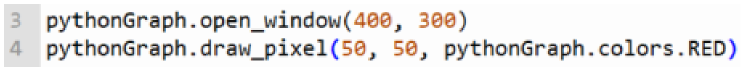
`draw_pixel(x, y, color)`

Description

Changes the color of a single pixel at location (`x, y`).

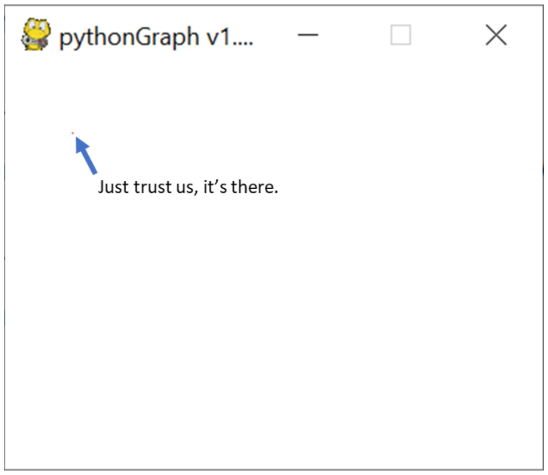
`color` can either be a predefined value (refer to `pythonGraph.colors`) or a custom color created using the `create_color` or `create_random_color` functions.

Example



```python
pythonGraph.open_window(400, 300)
pythonGraph.draw_pixel(50, 50, pythonGraph.colors.RED)
```

Output



Just trust us, it's there.

1.9 draw_text

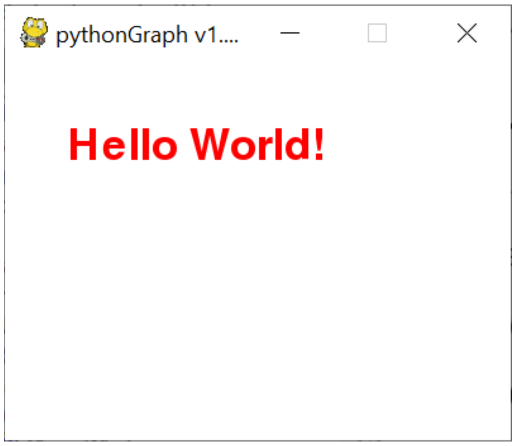
Usage
`draw_text(text, x, y, color, font_size)`

Description
Writes the specified text string to the pythonGraph window. `text` represents the string to be written. `(x,y)` denotes the coordinate of the top left corner of the string `color` can either be a predefined value (refer to `pythonGraph.colors`) or a custom color created using the `create_color` or `create_random_color` functions. `font_size` is an optional parameter that specifies the size of the text, in pixels. If this value is not provided, a default value of 30 will be used.

Example

```
pythonGraph.open_window(400, 300)
pythonGraph.draw_text("Hello World!", 50, 50, pythonGraph.colors.RED, 50)
```

Output



2. Mouse Operations

pythonGraph can determine the current location of the mouse. It can also determine whether or not a mouse click has occurred.

Before using these operations, please note that:

- `open_window` must be called first, otherwise a run-time error will occur.
- The window must be in focus. If the pythonGraph window is not on top, the user may have to click on it once before the application will respond to user mouse clicks.

METHODS DESCRIBED IN THIS CHAPTER

2.1 get_mouse_x and get_mouse_y

Usage

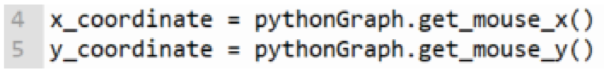
```
get_mouse_x()
get_mouse_y()
```

Description

These functions return the current **`x`** or **`y`** coordinate of the mouse.

Example

The following lines of code will store the mouse's current x and y coordinate in **`x_coordinate`**, and **`y_coordinate`**, respectively.



```
x_coordinate = pythonGraph.get_mouse_x()
y_coordinate = pythonGraph.get_mouse_y()
```

2.2 mouse_button_pressed

Usage

```
mouse_button_pressed(which_button)
```

Description

Returns **`True`** if the specified mouse button is clicked, and **`False`** otherwise. This function will only return **`True`** once per mouse click.

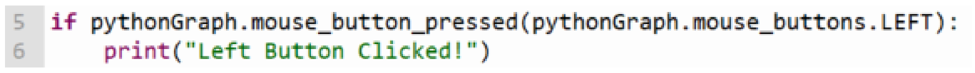
`which_button` can be one of the following values:

- **`pythonGraph.mouse_buttons.LEFT`**
- **`pythonGraph.mouse_buttons.CENTER`**
- **`pythonGraph.mouse_buttons.RIGHT`**

If the window is not on top, the user may have to click on it once before this function will be called.

Example

The following code snippet will print a string when the left mouse button is clicked:



```python
if pythonGraph.mouse_button_pressed(pythonGraph.mouse_buttons.LEFT):
    print("Left Button Clicked!")
```

2.3 mouse_button_down

Usage

```
mouse_button_down(which_button)
```

Description

Returns **`True`** if the specified mouse button is held down, and **`False`** otherwise.
Unlike **`mouse_button_pressed`**, this function will keep returning True for as long as the button is held down.

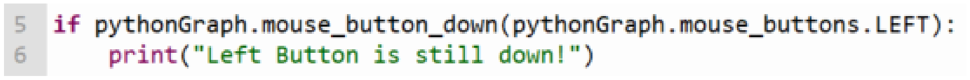
`which_button` can be one of the following values:

- **`pythonGraph.mouse_buttons.LEFT`**
- **`pythonGraph.mouse_buttons.CENTER`**
- **`pythonGraph.mouse_buttons.RIGHT`**

If the window is not on top, the user may have to click on it once before this function will be called.

Example

The following code snippet will print a string when the left mouse button is pressed:



```
if pythonGraph.mouse_button_down(pythonGraph.mouse_buttons.LEFT):
    print("Left Button is still down!")
```

2.4 mouse_button_released

Usage
`mouse_button_released(which_button)`

Description

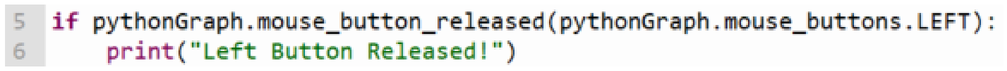
Returns **`True`** if the specified mouse button is released, and **`False`** otherwise.

`which_button` should be one of the following values:

- **`pythonGraph.mouse_buttons.LEFT`**
- **`pythonGraph.mouse_buttons.CENTER`**
- **`pythonGraph.mouse_buttons.RIGHT`**

Example

The following code snippet will print a string when the left mouse button is released:



```
if pythonGraph.mouse_button_released(pythonGraph.mouse_buttons.LEFT):
    print("Left Button Released!")
```

3. Keyboard Operations

These functions allow pythonGraph to determine if a keystroke has occurred.

Before using these operations, please note that:

- `open_window` must be called first, otherwise a run-time error will occur.
- The window must be in focus. If the pythonGraph window is not on top, the user may have to click on it once before the application will respond to user keyboard.

METHODS DESCRIBED IN THIS CHAPTER

- key_pressed
- key_down
- key_released

3.1 key_pressed

Usage

`key_pressed(which_key)`

Description

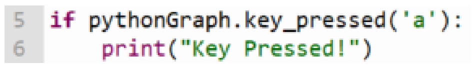
Returns **`True`** if the specified key is pressed, and **`False`** otherwise. This function will only return **`True`** once per keyboard press.

`which_key` is a lowercase string that represents the key that we want to check. For example:

- Letter Keys: ‘a’, ‘b’, ‘c’ . . . ‘z’
- Function Keys: ‘f1’, ‘f2’ . . .
- Arrow Keys: ‘up’, ‘down’, ‘left’, ‘right’
- Misc. Keys: ‘escape’, ‘numlock’, ‘*’

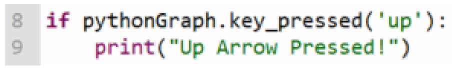
Example

The following code snippet will print a string when the ‘a’ key is pressed:



```python
if pythonGraph.key_pressed('a'):
    print("Key Pressed!")
```

The following code snippet will print a string when the up arrow key is pressed:



```python
if pythonGraph.key_pressed('up'):
    print("Up Arrow Pressed!")
```

3.2 key_down

Usage

```
key_down(which_button)
```

Description

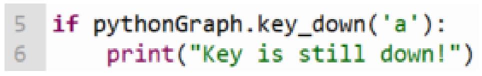
Returns **`True`** if the specified key is held down, and **`False`** otherwise. Unlike **`key_pressed`**, this function will keep returning True for as long as the key is held down.

`which_key` is a **lowercase** string that represents the key that we want to check. For example:

- Letter Keys: ‘a’, ‘b’, ‘c’ . . . ‘z’
- Function Keys: ‘f1’, ‘f2’ . . .
- Arrow Keys: ‘up’, ‘down’, ‘left’, ‘right’
- Misc. Keys: ‘escape’, ‘numlock’, ‘*’

Example

The following code snippet will repeatedly print a string for as long as the ‘a’ button is pressed:



```python
if pythonGraph.key_down('a'):
    print("Key is still down!")
```

3.3 key_released

Usage

`key_released(which_button)`

Description

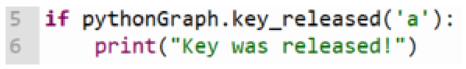
Returns **`True`** if the specified key is released, and **`False`** otherwise.

`which_key` is a **lowercase** string that represents the key that we want to check. For example:

- Letter Keys: ‘a’, ‘b’, ‘c’ . . . ‘z’
- Function Keys: ‘f1’, ‘f2’ . . .
- Arrow Keys: ‘up’, ‘down’, ‘left’, ‘right’
- Misc Keys: ‘escape’, ‘numlock’, ‘*’

Example

The following code snippet will print a string when ‘a’ key is released:



```python
if pythonGraph.key_released('a'):
    print("Key was released!")
```

4. Window Operations

The following functions allow pythonGraph to open, close, and update the graphics window.

METHODS DESCRIBED IN THIS CHAPTER

- open_window
- close_window
- get_window_height
- get_window_width
- set_window_title
- update_window
- window_closed
- window_not_closed

4.1 open_window

Usage

```
open_window(width, height)
```

Description

Creates a graphics window of the specified width and height (in pixels).

Important Notes:

- You can only have one pythonGraph window open at a time. If you attempt to open a second, an error will occur.
- The **`width`** and **`height`** dimensions cannot be negative

Example

The following code snippet will open a 400 x 300 pixel window:

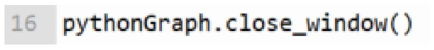
```
pythonGraph.open_window(400, 300)
```

4.2 close_window

Usage
`close_window(width, height)`

Description
Closes the pythonGraph window. A run-time error will occur if the graphics window is not open.

Example



```
pythonGraph.close_window()
```

4.3 get_window_height and get_window_width

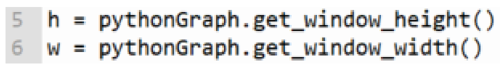
Usage

`get_window_height()`
`get_window_width()`

Description
Returns the height and width, respectively, of the window.

Example

The following snippet will store the window's height and width in the variables **`h`** and **`w`**, respectively.



```
h = pythonGraph.get_window_height()
w = pythonGraph.get_window_width()
```

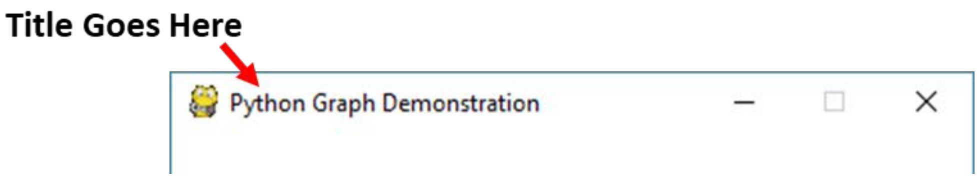
4.4 set_window_title

Usage

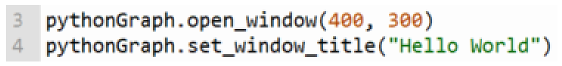
```
set_window_title(title)
```

Description

Changes the title of the pythonGraph window.

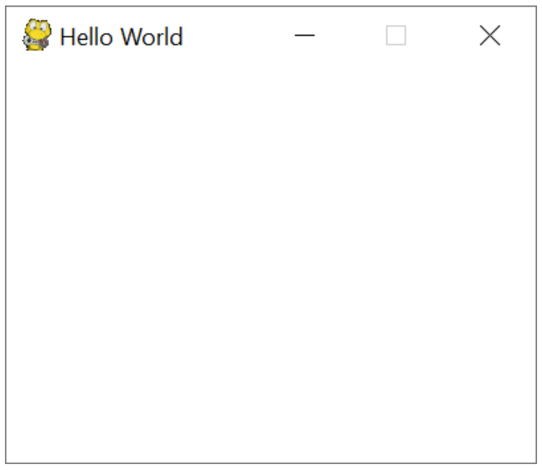


Example



```
pythonGraph.open_window(400, 300)
pythonGraph.set_window_title("Hello World")
```

Output



4.5 update_window

Usage

```
update_window(refresh_rate)
```

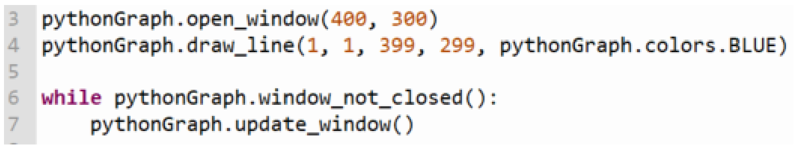
Description

Updates the visual contents of the pythonGraph window. All of the draw functions called prior to this will appear on the screen.

The refresh_rate is an optional parameter that specifies how much time, in milliseconds, the application should pause before executing the next line of code. This pause gives the user time to see what has been drawn. If this value is not provided, pythonGraph will use a default of 20ms.

Every pythonGraph program should call this function continuously (see example below). Without this call, the application will be nonresponsive.

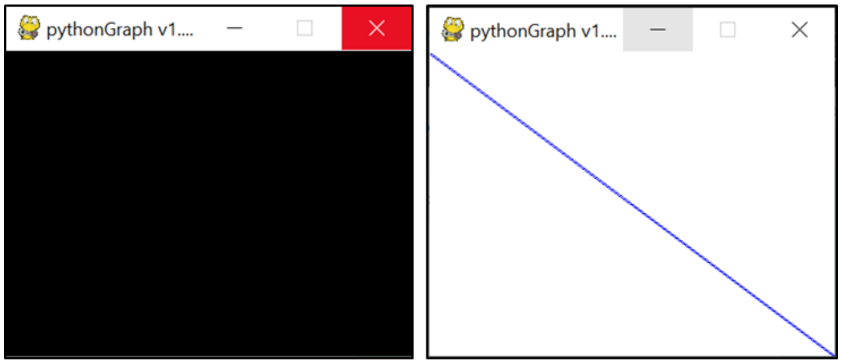
Example



```
pythonGraph.open_window(400, 300)
pythonGraph.draw_line(1, 1, 399, 299, pythonGraph.colors.BLUE)

while pythonGraph.window_not_closed():
    pythonGraph.update_window()
```

Output



Before `update_window()` After `update_window()`

4.6 delay

Usage

`delay(time)`

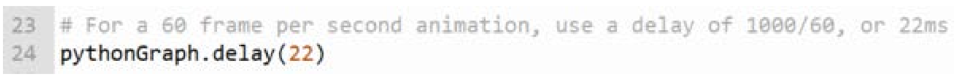
Description

Pauses the application for the specified amount of **`time`** (in milliseconds).

This function is typically called during an animation loop in order to allow the image to stay on the screen long enough for the user to see it.

The **`time`** parameter expects a positive integer.

Example



```python
# For a 60 frame per second animation, use a delay of 1000/60, or 22ms
pythonGraph.delay(22)
```

4.7 window_closed and window_not_closed

Usage

```
window_closed()
window_not_closed()
```

Description

`window_closed` returns **`True`** if the pythonGraph window has been closed by the user (*i.e.,* the user clicked on the ‘X’ in the top right corner), and **`False`** otherwise

`window_not_closed` does the opposite. It returns **`True`** if the pythonGraph window has been closed by the user (*i.e.,* the user clicked on the ‘X’ in the top right corner), and **`False`** otherwise

Example

The following code snippets opens a pythonGraph window, and continues to loop until the user clicks on the ‘X’ in the top right corner:

Using **`window_closed`**:

```python
pythonGraph.open_window(400, 300)

while not pythonGraph.window_closed():
    pythonGraph.update_window()
```

Using **`window_not_closed:`**

```python
pythonGraph.open_window(400, 300)

while pythonGraph.window_not_closed():
    pythonGraph.update_window()
```

5. Color Operations

pythonGraph comes with a predefined set of colors, as well as methods to easily generate custom and/or random colors as needed.

Predefined colors:

- `pythonGraph.colors.BLACK`
- `pythonGraph.colors.BLUE`
- `pythonGraph.colors.BROWN`
- `pythonGraph.colors.CYAN`
- `pythonGraph.colors.GRAY`
- `pythonGraph.colors.GREEN`
- `pythonGraph.colors.LIGHT_BLUE`
- `pythonGraph.colors.LIGHT_CYAN`
- `pythonGraph.colors.LIGHT_GRAY`
- `pythonGraph.colors.LIGHT_GREEN`
- `pythonGraph.colors.LIGHT_MAGENTA`
- `pythonGraph.colors.LIGHT_RED`
- `pythonGraph.colors.MAGENTA`
- `pythonGraph.colors.RED`
- `pythonGraph.colors.WHITE`
- `pythonGraph.colors.YELLOW`

Methods Described in this Chapter

- create_color
- create_random_color

5.1 create_color

Usage

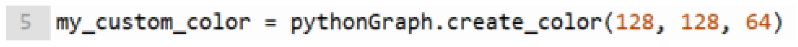
```
create_color(red, green, blue)
```

Description

Returns a color with the specified red, green, and blue combination.

`red`, **`green`**, and **`blue`** are all integer values between 0-255. Refer to https://www.colorspire.com/rgb-color-wheel/ to see how combinations of these three colors can be used to create other colors.

Example



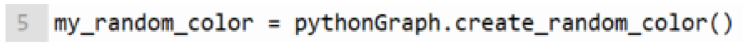
```
my_custom_color = pythonGraph.create_color(128, 128, 64)
```

5.2 create_random_color

Usage
`create_random_color()`

Description
Returns a color with a random red, green, and blue combination.

Example



```
my_random_color = pythonGraph.create_random_color()
```

6. Music Operations

pythonGraph provides limited functions to play sound effects and background music. A sound effect is defined as a short sound clip (< 1s). Background music, in contrast, can range from seconds to minutes, and can be set to be played once or on a continuous loop.

WAV and MP3 files are currently supported.

METHODS DESCRIBED IN THIS CHAPTER

- play_sound_effect
- play_music
- stop_music

6.1 play_sound_effect

Usage

```
play_sound_effect(filename)
```

Description

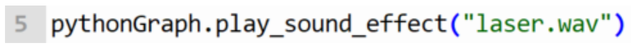
Plays the specified sound file once, if a channel is available.

The **`filename`** parameter specifies where the file to be played is located on the computer. Typically, your sound effect files should be in the same folder as your python application.

This method supports WAV files. The larger the file, the longer it will take for the application to load and play it.

Example

This snippet will play the sound "sound.wav", assuming that the file is in the same folder.



```
pythonGraph.play_sound_effect("laser.wav")
```

6.2 play_music

Usage

```
play_music(filename, loop)
```

Description

Plays the specified music file, if a channel is available.

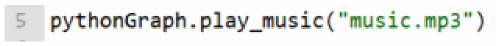
The `filename` parameter specifies where the file to be played is located on the computer. Typically, your music files should be in the same folder as your python application.

The `loop` parameter is optional, and specifies whether or not to play the music on a continuous loop. By default, this value is set to `True`.

This method supports WAV and MP3 files. The larger the file, the longer it will take for the application to load and play it.

Example

This snippet will play the sound “music.mp3”, assuming that the file is in the same folder.



```
pythonGraph.play_music("music.mp3")
```

6.3 stop_music

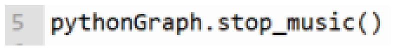
Usage

```
stop_music()
```

Description

Stops any music that is currently playing. This function can be safely called, even if music is not playing.

Example



```
pythonGraph.stop_music()
```