Nicholas Bowman, Sonja Johnson-Yu, Kylie Jue
CS 106AP

Handout #14
July 24, 2019

Campy Reference Guide

Last updated on August 12, 2019.

NOTE: This reference guide is meant to act as a resource for Assignments 5 and 6. You are not allowed to use the campy library for Assignment 4, which is meant to give you practice with basic Tkinter.

Developed by Sam Redmond, campy is a partial rewrite of Stanford's ACM libraries for introductory programming, specifically in Python. In CS106AP this quarter, we will be using campy's graphics and gui modules, which are built on top of Tkinter, to create our graphics programs!

The first part of this handout will explain how to install campy, and the second part will include more in-depth documentation of the campy modules we'll be using. When applicable, we will also link to specific pages in the more extensive, online campy documentation for your reference.

> **NOTE:** We will be updating this guide throughout the quarter as we continue to expand our usage of the campy library (for example, between Assignments 5 and 6). While we will make announcements about important changes, you can check the "last updated" date in red at the top of the handout to see when the most recent version was created.

TABLE OF CONTENTS

Installation

In the terminal commands that follows, you should use **`py`** instead of **`python3`** if you are working on a Windows computer.

To install campy, we will use pip, just like we did with the SimpleImage library. Run the following two commands in your terminal application:

```
$ python3 -m pip install --upgrade pip
$ python3 -m pip install campy
```

The first command upgrades your version of pip, and the second downloads the latest version of campy. To confirm you've installed campy correctly, run the following command:

```
$ python3 -m pip show campy
```

You should see an output that looks like Figure 1. In particular, note that the version of campy you are using should be **`0.0.1.dev19`**.

Figure 1: You should see the above information about campy in your terminal application after installing the library and running **`` `python3 -m pip show campy` ``**.

Updating campy

If you ever want to update campy when new versions are released, you can run the following command:

```
$ python3 -m pip install --upgrade campy
```

We will let you know if there are any mandatory campy updates that we'll need you to download, but in general, it's also a good practice to periodically update the libraries you have installed. Doing so ensures that the modules stay bug-free and remain compatible with other updates on your computer.

Uninstalling and reinstalling campy

If for some reason you run into issues with updating campy or just need to uninstall and reinstall the library, you can run the following commands in your terminal application:

```
$ python3 -m pip uninstall campy
$ python3 -m pip install campy
```

This completely removes your existing version of campy and replaces it with the most up-to-date version of the library.

Module documentation

Campy graphics use the same canvas paradigm that we saw with Tkinter: graphics get drawn on a canvas (called a **`GWindow`**, or graphics window), with graphical elements that are added later layered on top of graphical elements that were added earlier (see Figure 2). In other words, the canvas acts like a real canvas where newly added layers of paint overlay previous layers.

Campy also provides GUI-related features that determine when our code executes. These are all located in the **`gui`** package and include both timer functions and also modules that give us easy access to user "events," such as mouse movements, button clicks, etc.

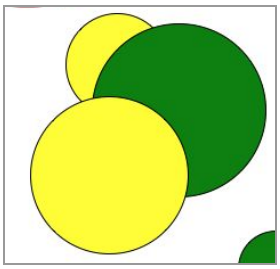

Figure 2: The small yellow circle in the upper left was added to the canvas first, then the large green circle, and lastly the second larger yellow circle. This is consistent with Tkinter's canvas paradigm.

Importing campy modules

We will primarily be using the **`graphics`** package and the **`gui`** package from the campy libraries. Each of these packages contains additional packages and module files that we import for you on an assignment-by-assignment basis.

Inspect the import statements at the top of your code files to see examples of how to use different campy features and to better understand where each module is located (within what packages). Import statements take on the following format:

```
from campy.package.subpackage.module import function, Class, Class2
```

In the **`from`** portion of each import statement, nested subpackages are separated by a period **`(.)`**, and there can be multiple nested **`subpackage`**s, depending on where the **`module`** is located. **`module`** will match the name of the module's Python file within campy, and each **`package`** or **`subpackage`** matches a folder name. We determine which **`function`**s and **`Class`**es (more on Python classes in Lectures 19 through 22!) we want to **`import`** from the specified **`module`** based on what campy features we want to use.

We will always provide all of the imports you need for each assignment/sample code file, but we've provided the above information for folks who are interested in understanding how importing the modules works (and so that you can implement your own files from scratch if you'd like!). Feel free to add supplementary import statements in your programs if you choose to extend their functionality.

Creating a GWindow (canvas)

To start working on a graphics program, the first step is to always create a canvas, called a **`GWindow`** in campy. You'll want to store the canvas in a variable so that you can access it later and continue to add other graphics objects to it:

```
window = GWindow(width=100, height=100, title='Breakout')
```

When creating a **`GWindow`**, you aren't required to pass in any arguments (their defaults can be found here), but the common keyword arguments that we will be using are **`width`**, **`height`**, and **`title`**. The width and height of the window each default to 500, while the title displayed at the top of the GUI window defaults to the empty string.

Once you've created your window, you'll want to keep the following properties and functions in mind:

`window.width`	A property for accessing the window's width
`window.height`	A property for accessing the window's height
`window.add()`	Add a specified object to the window. Can optionally specify the x and y coordinates to place the object add
`window.remove()`	Removes a specified object from the window
`window.clear()`	Clears all content from window and returns it to its blank state
`window.get_object_at()`	Gets the object (if any) located at a given location in the window

Figure 3: Common **`GWindow`** properties and functions

Full **`GWindow`** documentation (all properties, functions, and default kwarg values) can be found here.

Adding GObjects to the canvas

Once you've created your **`GWindow`**, you'll want to add different **`GObject`**s to it. **`GObject`**s are the many different kinds of shapes that can be created using campy. Every

GObject has the following properties and functions:

obj.width	A property for accessing the object's width
obj.height	A property for accessing the object's height
obj.x	A property for accessing the object's x-coordinate
obj.y	A property for accessing the object's x-coordinate
obj.color	A property for accessing the object's outline color. A list of colors supported by campy can be found here.
obj.fill_color	A property for accessing the object's interior (fill) color. A list of colors supported by campy can be found here. [**GRect** and **GOval** only]
obj.filled	A property for accessing whether or not the object is filled in (either True or False) [**GRect** and **GOval** only]
obj.move(dx,dy)	Moves the object on the screen using the specified displacements

Figure 4: Common **GObject** properties and functions

Full **GObject** documentation can be found here. The primary types of **GObject**s that we will be using are seen in Figure 5. You can call any of the above **GObject** functions on these object types as well, and the table shown on the following page demonstrates how to create each of the shapes using the correct parameters.

GRect	**rectangle = GRect(width, height, x=0, y=0)** where **(x, y)** is the upper left corner of the rectangle
GOval	**oval = GOval(width, height, x=0, y=0)** where **(x, y)** is the upper left corner of the bounding rectangle around the oval
GLine	**line = GLine(x0, y0, x1, y1)** where **(x0, y0)** and **(x1, y1)** are the starting and ending points for the line
GLabel	**text_label = GLabel(label, x=0, y=0)** where **label** is the string contained inside the text label and **(x, y)** is the bottom left corner of the label

Figure 5: Four types of **GObject**s that we will be using in our programs. Each is its own data type and may have additional properties and functions that can be found in the corresponding linked documentation in the leftmost column.

Once you've created the **GObject** you want to use, make sure to add it to your **GWindow** using **window.add(object)**, where object is the **GObject** you've just declared!

Creating an event loop for continuous movement

When creating graphics programs, in particular for graphical games, we will often want objects in the **`GWindow`** to have continuous movement. This can be achieved by creating an **event loop**.

Within the event loop, you will write the code that causes your objects to move a small amount at each timestep and you will use the **`timer`** function **`pause()`** to determine how many times per second your loop should repeat. Although event loops can use either for loops or while loops (depending on whether or not you know how many times you want the animation to repeat), the most common paradigm is to use a while loop.

The **`timer`** module includes additional functions that manipulate when your code executes, and you can find its full documentation here.

Below is a sample event loop that animates the movement of a rectangle from the left side of the window to the right side of the window. Notice the event loop has three important components: a **`while`** loop that continuously updates the position of the rectangle, a conditional statement that eventually terminates the loop, and a **`pause()`** call that creates the "animation" effect by controlling how much time passes between **`while`** loop iterations.

```
def animate_rect_movement(window):
      rect = GRect(30, 40)
      window.add(rect)
      while True:
            if rect.x + rect.width >= window.width:
                  break
            rect.move(DELTA_X, 0)
            pause(1000 / 120) # 120 updates per second)
```

Detecting mouse events

In order to create graphics programs that are interactive, we need to be able to respond to user input. Campy's **`mouse`** module makes it possible for us to detect different forms of mouse events from the user. In particular, we can detect four types of mouse movements:

`onmouseclicked(fn)`	Occurs when the user **presses and then releases** any button on their mouse
`onmousereleased(fn)`	Occurs when the user **releases** any button on their mouse
`onmousemoved(fn)`	Occurs when the user **moves** the mouse in any direction
`onmousedragged(fn)`	Occurs when the user **both presses and moves** their mouse

Figure 6: The five different types of mouse events that campy can detect.

To use mouse events effectively, you must

- First create a **mouse listener** function (represented as **`fn`** in Figure 6) that will get called when the corresponding mouse event occurs
- Define the mouse listener function to take in an **`event`** parameter. This allows campy to recognize it as a valid mouse listener function and also gives you access to the mouse event's **`x`** and **`y`** coordinates within your function (see example below).
- Pass your function in as an argument to one of the mouse event functions listed in Figure 6, depending on which event you would like it to correspond to. **Note that when passing your function as an argument, you leave out the trailing parentheses.**

The code below demonstrates how you would write a program that occurs every time you click your mouse within the **`GWindow`**. The program prints "Hello" along with the location in the window where the click occurred. For example, if you clicked at point (x=5, y=25) in the **`GWindow`**, you would see the message "Hello from (5, 25)" appear in your console.

```
def say_hello(event):
    print('Hello from (', str(event.x), ', ', str(event.y), ')')

def main():
    window = GWindow()
    onmouseclicked(say_hello)
```

Creating GImages

To enable **`GImage`** functionality in your program, make sure to import **`campy.graphics.gimage`** at the top of your file. To create a **`GImage`**, you should use the following function:

```
image = GImage.from_file(filename_string)
```

`GImage.from_file()` returns a **`GImage`** object that is generated from the filename argument passed in as a string. Once you've created your **`GImage`**, the object behaves very similarly to a **`SimpleImage`** object that is also a **`GObject`**!

To add your **`GImage`** to the **`GWindow`**, you should do the same as you would with any other **`GObject`** and use **`window.add(image, x, y)`**.

You can manipulate **`GImage`**s similarly to how you did with **`SimpleImage`**s. You should make sure that you still have the Pillow library installed in order for **`GImage`**s to work correctly. **Additionally, you must add the `GImage` to the window before attempting any pixel manipulations.**

Specifically, the following attributes exist for **`GImage`**s:

- **`image.height`**
- **`image.width`**

- `image.get_pixel(x, y)`
- `image.set_pixel(x, y, pixel)`

Note that unlike **`SimpleImage`**s, **`GImage`**s currently **do not support** image resizing, for each loops, or the creation of blank images. In addition, graphics **`Pixel`** objects are not associated with a specific **`GImage`** so in order to edit a pixel within an image, you need to first use **`get_pixel()`** to get the pixel and then use **`set_pixel()`** to update the pixel within the image. This also means that **`Pixel`** objects do not have **`x`** and **`y`** attributes.

However, you can iterate over all of the pixels in an image using a double for range() loop pattern to access or manipulate the pixel RGB values:

```
image = GImage.from_file("image.png")
window.add(image)
for y in range(image.height):
    for x in range(image.width):
        # Use pixel = image.get_pixel(x, y) to get the pixel
        # Use pixel.red, pixel.green, pixel.blue to modify it
        # Use image.set_pixel(x, y, pixel) to update the pixel
        # within the image
```

You can also use the following **`GObject`** methods and attributes on **`GImage`**s:

`obj.width`	A property for accessing the object's width
`obj.height`	A property for accessing the object's height
`obj.x`	A property for accessing the object's x-coordinate
`obj.y`	A property for accessing the object's x-coordinate
`obj.move(dx,dy)`	Moves the object on the screen using the specified displacements

Figure 7: The **`GObject`** methods that also apply to **`GImage`**s. You should replace **`obj`** with our **`GImage`** object (or the variable that is storing it).

Appendix

Colors

Here is a table of all the colors that campy supports:

`'aliceblue'`	`'antiquewhite'`	`'aqua'`	`'aquamarine'`	`'azure'`
`'beige'`	`'bisque'`	`'black'`	`'blanchedalmond'`	`'blue'`
`'blueviolet'`	`'brown'`	`'burlywood'`	`'cadetblue'`	`'chartreuse'`
`'chocolate'`	`'coral'`	`'cornflowerblue'`	`'cornsilk'`	`'crimson'`
`'cyan'`	`'darkblue'`	`'darkcyan'`	`'darkgoldenrod'`	`'darkgray'`

'darkgreen'	'darkgrey'	'darkkhaki'	'darkmagenta'	'darkolivegreen'
'darkorange'	'darkorchid'	'darkred'	'darksage'	'darksalmon'
'darkseagreen'	'darkslateblue'	'darkslategray'	'darkslategrey'	'darkturquoise'
'darkviolet'	'deeppink'	'deepskyblue'	'dimgray'	'dimgrey'
'dodgerblue'	'firebrick'	'floralwhite'	'forestgreen'	'fuchsia'
'gainsboro'	'ghostwhite'	'gold'	'goldenrod'	'gray'
'green'	'greenyellow'	'grey'	'honeydew'	'hotpink'
'indianred'	'indigo'	'ivory'	'khaki'	'lavender'
'lavenderblush'	'lawngreen'	'lemonchiffon'	'lightblue'	'lightcoral'
'lightcyan'	'lightgoldenrodyellow'	'lightgray'	'lightgreen'	'lightgrey'
'lightpink'	'lightsage'	'lightsalmon'	'lightseagreen'	'lightskyblue'
'lightslategray'	'lightslategrey'	'lightsteelblue'	'lightyellow'	'lime'
'limegreen'	'linen'	'magenta'	'maroon'	'mediumaquamarine'
'mediumblue'	'mediumorchid'	'mediumpurple'	'mediumseagreen'	'mediumslateblue'
'mediumspringgreen'	'mediumturquoise'	'mediumvioletred'	'midnightblue'	'mintcream'
'mistyrose'	'moccasin'	'navajowhite'	'navy'	'oldlace'
'olive'	'olivedrab'	'orange'	'orangered'	'orchid'
'palegoldenrod'	'palegreen'	'paleturquoise'	'palevioletred'	'papayawhip'
'peachpuff'	'peru'	'pink'	'plum'	'powderblue'
'purple'	'red'	'rosybrown'	'royalblue'	'saddlebrown'
'sage'	'salmon'	'sandybrown'	'seagreen'	'seashell'
'sienna'	'silver'	'skyblue'	'slateblue'	'slategray'
'slategrey'	'snow'	'springgreen'	'steelblue'	'tan'
'teal'	'thistle'	'tomato'	'turquoise'	'violet'
'wheat'	'white'	'whitesmoke'	'yellow'	'yellowgreen'