Graphics 2.0

CS106AP Lecture 18
Day 1!

Roadmap

- Programming Basics
- The Console
- Images
- Data structures
- Midterm
- Object-Oriented Programming
- Everyday Python
- Graphics
  - Graphics 1.0
  - Graphics 2.0
  - Event-driven programming

Life after CS106AP!
Today’s questions

How can we more easily build complex graphical programs?

How can we get our programs to produce random output?
Today’s topics

1. Review
2. Graphics Libraries: Campy
3. Randomness
4. What’s next?
Review
How can we benefit from others’ code and allow others to use our code?

Modules!
Definition

modules

*.py* files containing working code for reuse across programs; sometimes also called “libraries”
Definition

package
A directory of Python module(s)
Using modules

1. Import the module

   \texttt{import module}

2. Use the predefined functions!

   \texttt{module.function()}

\textit{We call this building your code “on top of” the module.}
How can we create graphical programs?

GUI libraries (modules)!
**Definition**

**Graphical User Interface (GUI)**
Visual elements that you interact with within applications (windows, buttons, scroll bars, etc.)
GUI libraries

- Windows, Mac OS, Linux each have their own GUI systems

- **GUI libraries** are modules that support the GUI (i.e. their functions allow you to interact with your computer’s GUI system)

- **tkinter** is Python’s standard GUI package
  - Many other GUI libraries are built on top of **tkinter**!
    - (more on this later this week)
Tkinter fundamentals

- `import tkinter`

- Everything gets drawn on a `canvas`
  - In today’s examples and Assignment 4, we’ll provide the code for creating the canvas.

- When dealing with x, y coordinates on the canvas, Tk converts floats to integers internally to address pixels
  - **Note**: Python floats aren’t exact!
The Shocking Truth about Floats
The Shocking Truth about Floats

0.2 & 0.1’s only sum: Floating point number
The Shocking Truth about Floats

- Floats aren’t precise!
- This is a result of how they’re stored in computer memory
  - Same issue across languages
- Avoid ==
Our Use of Floats Yesterday

\[
\text{fraction} = \frac{i}{n - 1}
\]
\[
x_2 = (\text{width} - 1) \times \text{fraction}
\]

compared to

\[
i \times \left(\frac{(\text{width} - 1)}{n - 1}\right)
\]

e.g. \(i = 22, n = 23, \text{width} = 101\)
The Shocking Truth about Floats

I can do any calculation in Python

You know floats are imprecise, right?

>>> 0.1 + 0.1 + 0.1
0.3000000000000004
Tkinter canvas functions

- `canvas.create_line(x1, y1, x2, y2)`

- `canvas.create_oval(x1, y1, x2, y2)`

- `canvas.create_text(x, y, text='hi', anchor=tkinter.NW)`
Tkinter canvas functions

- `canvas.create_line()`
- `canvas.create_oval()`
- `canvas.create_text(x, y, text='hi', anchor=tkinter.S)`

The anchor argument determines where \((x, y)\) is. It defaults to `tkinter.CENTER`.
Tkinter canvas functions

- `canvas.create_line()`
- `canvas.create_oval()`
- `canvas.create_text(x, y, text='hi', anchor=tkinter.S)`

It can be any of `[tkinter.]N, S, E, W, NW, SW, NE, SW, or CENTER.`
Tkinter canvas functions

- `canvas.create_line()`
- `canvas.create_oval()`
- `canvas.create_text(x, y, text='hi', anchor=tkinter.S)`

It can be any of `[tkinter.]N, S, E, W, NW, SW, NE, SW, or CENTER.

They are constants inside the tkinter module!
Tkinter canvas functions

- `canvas.create_line()`
- `canvas.create_oval()`
- `canvas.create_text(x, y, text='hi', anchor=tkinter.S)`

keyword arguments!
**Definition**

**keyword arguments (kwargs)**
Arguments whose names matter but whose positions do not
Keyword (named) arguments: **kwargs

- Come *after* the unnamed **positional** arguments whose order matter

**Definition**

**positional arguments**
Unnamed arguments whose order matter
Keyword (named) arguments: *kwargs

- Come after the unnamed positional arguments whose order matter
- Name matters, but order doesn’t matter
- Help make function calls clearer
- Enable default values for parameters (make arguments optional)
Keyword (named) arguments: **kwargs

def calculate_bill(cost, tip=0.15, tax=0.0725):
    return (1 + tax) * (1 + tip) * cost
Keyword (named) arguments: **kwargs

def calculate_bill(cost, tip=0.15, tax=0.0725):
    return (1 + tax) * (1 + tip) * cost
Keyword (named) arguments: **kwargs

```python
def calculate_bill(cost, tip=0.15, tax=0.0725):
    return (1 + tax) * (1 + tip) * cost

>>> calculate_bill(10)
```

Keyword (named) arguments: **kwargs

def calculate_bill(cost, tip=0.15, tax=0.0725):
    return (1 + tax) * (1 + tip) * cost

>>> calculate_bill(10)

we don’t have to pass in arguments that have default values!
Keyword (named) arguments: `kwargs`

```python
def calculate_bill(cost, tip=0.15, tax=0.0725):
    return (1 + tax) * (1 + tip) * cost

>>> calculate_bill(10)
12.333749999999998
```

We don’t have to pass in arguments that have default values!
Keyword (named) arguments: **kwargs

def calculate_bill(cost, tip=0.15, tax=0.0725):
    return (1 + tax) * (1 + tip) * cost

>>> calculate_bill(10, tax=0)
Keyword (named) arguments: *kwargs

```python
def calculate_bill(cost, tip=0.15, tax=0.0725):
    return (1 + tax) * (1 + tip) * cost

>>> calculate_bill(10, tax=0)
11.50
```
Keyword (named) arguments: \texttt{kwargs}

```python
def calculate_bill(cost, tip=0.15, tax=0.0725):
    return (1 + tax) * (1 + tip) * cost

>>> calculate_bill(10, tax=0)
11.50
```

we can specify a value for our \texttt{kwargs}.
Keyword (named) arguments: **kwargs

```python
def calculate_bill(cost, tip=0.15, tax=0.0725):
    return (1 + tax) * (1 + tip) * cost

>>> calculate_bill(tax=0, 10)
```
Keyword (named) arguments: `kwargs`

def calculate_bill(cost, tip=0.15, tax=0.0725):
    return (1 + tax) * (1 + tip) * cost

>>> calculate_bill(tax=0, 10)
SyntaxError
Keyword (named) arguments: **kwargs

```python
def calculate_bill(cost, tip=0.15, tax=0.0725):
    return (1 + tax) * (1 + tip) * cost

>>> calculate_bill(tax=0, 10)
```

**kwargs must follow positional args!
Keyword (named) arguments: **kwargs

def calculate_bill(cost, tip=0.15, tax=0.0725):
    return (1 + tax) * (1 + tip) * cost

>>> calculate_bill(10, 0.15, 0)
Keyword (named) arguments: `kwargs`

```python
def calculate_bill(cost, tip=0.15, tax=0.0725):
    return (1 + tax) * (1 + tip) * cost

>>> calculate_bill(10, 0.15, 0)
```

`this will behave just like passing in positional args (cost, tip, tax)`
draw_pyramid()
Note about graphics in this class

• You have now learned everything you need for BabyNames
• Do not use the following material on BabyNames!
  ○ We will use it in subsequent assignments!
What can we build with graphical programs?
What can we build with graphical programs?

[DEMO]
How can we more easily build complex graphical programs?
How can we more easily build complex graphical programs?

Leverage others’ code: other graphics libraries!
Other Graphics Library: Campy

• Python version of Stanford’s ACM Graphics Libraries
  ○ by former SL Sam Redmond!
• Allows us to easily create graphical objects and add them to a canvas
• Built on top of tkinter!
How can I leverage Campy in my programs?

- Import modules from the campy library
- New import syntax:

```python
from campy.graphics.gobjects import GOval
```
How can I leverage Campy in my programs?

- Import modules from the campy library
- New import syntax:

```python
from campy.graphics.gobjects import GOval
```

- **path to specific module**
- **object (could be a function)**
How can I leverage Campy in my programs?

- Import modules from the campy library
- New import syntax:

```python
from campy.graphics.gobjects import GOval
```

You’ve seen this before:

```python
from simpleimage import SimpleImage
```
Using modules - two ways

1. Import the module
   
   ```
   import module
   ```

2. Use the predefined functions!
   
   ```
   module.function()
   ```

   *Method A*
Using modules - two ways

1. Import the module
   ```python
   import module
   ```
2. Use the predefined functions!
   ```python
   module.function()
   ```

   *Method A*

1. Import the function/object from module
   ```python
   from module import function
   ```
2. Use the function/object!
   ```python
   function()
   ```

   *Method B*
Using modules - two ways

1. Import the module

   ```python
   import campy.graphics.gobjects
   ```

2. Use the predefined functions/objects!

   ```python
   campy.graphics.gobjects.GOval(width, height)
   ```

   *Method A*
Using modules - two ways

1. Import the function/object from module
   
   ```python
   from campy.graphics.gobjects import GOval
   ```

2. Use the function/object!

   ```python
   GOval(width, height)
   ```

   *Method B*
Using modules - two ways

1. Import the function/object from module
   ```python
   from campy.graphics.gobjects import GOval, GRect
   ```

2. Use the function/object!
   ```python
   GOval(width, height)
   ```

**Method B**

You can import multiple things!
Using modules - two ways

1. Import the module

   import module

2. Use the predefined functions!

   module.function()

   \textit{Method A}

---

1. Import the function/object from module

   from module import function

2. Use the function/object!

   function()

\textit{Method B}
Using modules - two ways

1. Import the module
   
   ```python
   import module
   ```

2. Use the predefined functions!
   
   ```python
   module.function()
   ```

   **Method A**

1. Import the function/object from module
   
   ```python
   from module import function
   ```

2. Use the function/object!
   
   ```python
   function()
   ```

   **Method B**

   more concise, especially if you're only importing 1-2 things
Why bother with Campy?

● Tkinter can be annoying to use
  ○ Code can become extremely long
● Using other libraries (like Campy) is more convenient
Campy’s Graphical Toolkit

- GRect
- GOval
- GLine
- GLabel
- GWWindow
def get_rect():
    rect = GRect(width=100, height=50, x=25, y=25)
    rect.filled = True
    rect.fill_color = 'blue'
    return rect
GRect

def get_rect():
    rect = GRect(width=100, height=50, x=25, y=25)
    rect.filled = True
    rect.fill_color = 'blue'
    return rect
GRect

def get_rect():
    rect = GRect(width=100, height=50, x=25, y=25)
    rect.filled = True
    rect.fill_color = 'blue'
    return rect

(x, y) in top left corner!
def get_oval():
    oval = GOval(width=100, height=50, x=0, y=0)
    oval.filled = True
    oval.fill_color = 'green'
    return oval
def get_oval():
    oval = GOval(width=100, height=50, x=0, y=0)
    oval.filled = True
    oval.fill_color = 'green'
    return oval

(x, y) in top left corner!
GLine

```python
line = GLine(x0, y0, x1, y1)
```

# similar to tk’s create_line() function!
GLabel

label = GLabel(text, x, y)

# similar to tk’s create_text() function!

label = GLabel('hi', 0, 0)
GLabel

label = GLabel(text, x, y)
# similar to tk’s create_text() function!
label = GLabel(‘hi’, 0, 0)

Note: coordinates for GLabel are bottom left
Campy's Graphical Toolkit

- GRect
- GOval
- GLine
- GLabel
- GWindow
Getting information from GObjects

`obj.width`

`obj.height`

`obj.x`

`obj.y`

`obj.filled`

`obj.fill_color`
A small problem...

We know how to make GObjects, but we don’t know how to display them!
Campy’s Graphical Toolkit

- GRect
- GOval
- GLine
- GLabel
- GWWindow
GWindow

We know how to make GObjects, but we don’t know how to display them!
GWindow

We know how to make GObjects, but we don’t know how to display them!

- GWindow is Campy’s version of the canvas that we used in tkinter
GWindow

We know how to make GObjects, but we don’t know how to display them!

- GWindow is Campy’s version of the canvas that we used in tkinter

```python
window = GWindow(height, width, title)
```
GWindow

We know how to make GObjects, but we don’t know how to display them!

- GWindow is Campy’s version of the canvas that we used in tkinter

```python
window = GWindow(height, width, title)
```

specifies the dimensions of the window
GWindow

We know how to make GObjects, but we don’t know how to display them!

- GWindow is Campy’s version of the canvas that we used in tkinter

```python
window = GWindow(height, width, title)
```

- optional argument, will create title at top of window
- specifies the dimensions of the window
window = GWindow(height=400, width=600)
GWindow - Add

window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50, x=25, y=25)
GWindow - Add

window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50, x=25, y=25)
window.add(rect)
window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50, x=25, y=25)
window.add(rect)

.add() displays the object in
the window!
GWindow - Add

window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)
GWindow - Add

window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)

note we're not specifying x, y when we create the GRect
window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)
window.add(rect, x=25, y=25)

note we’re not specifying x, y when we create the GRect
GWindow - Add

window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)
window.add(rect, x=25, y=25)

note we’re not specifying x, y when we create the GRect

specify x, y in add()
GWindow - Add

```python
window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)
window.add(rect, x=25, y=25)
```
window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)
window.add(rect, x=25, y=25)
window.remove(rect)
window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)
window.add(rect, x=25, y=25)
window.remove(rect)
window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)
window.add(rect, x=25, y=25)
GWindow - Clear

window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)
window.add(rect, x=25, y=25)
oval = GOval(25, 25)
GWindow - Clear

window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)
window.add(rect, x=25, y=25)

oval = GOval(25, 25)
window.add(oval, x=0, y=0)
GWindow - Clear

window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)
window.add(rect, x=25, y=25)

oval = GOval(25, 25)
window.add(oval, x=0, y=0)

window.clear()
window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)
window.add(rect, x=25, y=25)
oval = GOval(25, 25)
window.add(oval, x=0, y=0)
window.clear()
# remove all objects from the window!
GWindow - Get Object At

window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)
window.add(rect, x=25, y=25)
window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)
window.add(rect, x=25, y=25)
obj = window.get_object_at(40, 50)
GWindow - Get Object At

window = GWindow(height=400, width=600)
rect = GRect(width=100, height=50)
window.add(rect, x=25, y=25)
obj = window.get_object_at(40, 50)
print(obj == rect)  # True
GWindow - Summary

- The GWindow stores all the information about the objects in the window!
- GObjects are added in order (most recently added are on top)

```python
window = GWindow(height, width, title)
window.add(obj, x, y)  # x, y are optional
window.remove(obj)
window.clear()  # remove all objects
window.get_object_at(x, y)  # returns topmost object
```
DEMO: Draw a simple snowman!
How can we get our programs to produce random output?
A Random Note
Randomness

- It is impossible for a computer to generate a sequence of numbers that is truly random.
Randomness

- It is impossible for a computer to generate a sequence of numbers that is truly random
  - Computers are deterministic and designed to eliminate randomness!
Randomness

- It is impossible for a computer to generate a sequence of numbers that is truly random
  - Computers are deterministic and designed to eliminate randomness!
  - We can approximate randomness through “pseudo-random numbers”
Randomness

- It is impossible for a computer to generate a sequence of numbers that is truly random
  - Computers are deterministic and designed to eliminate randomness!
  - We can approximate randomness through “pseudo-random numbers”
- It’s valuable to have randomness in programs, either for cool effects or for conducting statistical tests.
The `random` module

- `random` is a useful Python module
The `random` module

- `random` is a useful Python module
- We saw it yesterday!
The `random` module

- `random` is a useful Python module
- We saw it yesterday!

```python
import random

COLORS = ['red', 'orange', 'yellow', 'green', 'blue']

color = random.choice(COLORS)
```
The **random** module

- **random** is a useful Python module
- We saw it yesterday!

```python
color = random.choice(COLORS)
```

randomly selects from a list
The **random** module

- Generate a random float $0 \leq x < 1$

  `random.random()`

- Generate a random int $a \leq x \leq b$

  `random.randint(a, b)`

- Randomly select from a list

  `random.choice(lst)`

*note that it is inclusive!*
The `random` module

- Set up a random number generator
  - This makes the output consistent between runs
  - This is very useful for testing random programs!

`random.seed(n)`
What’s next?