#1-3: Function Calls

CS SCHOLARS – PROGRAMMING

Learning Goals

Identify the **argument(s)** and **returned value** of a function call

Use **libraries** to import functions in categories like math and randomness

Use the **graphics library** to construct images algorithmically

Repeating Actions is Messy

Sometimes we want to perform the same algorithm many times on different inputs.

For example, say we want to personalize a young child's reading material so that it uses their pet's name.

We could copy and paste the first bit of code, then change the necessary parts. But if we're sloppy this might cause errors.

```
pet1 = "Spot"
pet2 = "Stella"
pet3 = "Kimchee"

print("See " + pet1 + ". See " + pet1 +
      " run. Run, " + pet1 + ", run!")

print("See " + pet2 + ". See " + pet2 +
      " run. Run, " + pet2 + ", run!")

print("See " + pet3 + ". See " + pet1 +
      " run. Run, " + pet3 + ", run!")
```

Functions Represent Abstract Actions

A better approach is to put the core action being repeated into a **function**.

A function is a code construct that represents an algorithm. We can **define** a function once, then **call** it many times.

We can also use functions that have already been defined by Python.

Function Calls

Call Functions with Parentheses

We've already seen how to call a function on a specific input, because `print` is just a function! This is done using **parentheses**.

`functionName(input1, input2, ...)`

The number of inputs provided inside the parentheses depends on how many inputs the function expects. Each input should be an **expression**.

A Few New Functions

To help us explore how functions work, let's introduce a few new functions. These are **built-in functions**, like `print`; that means we can call them in Python directly.

```
abs(-2) # 2; absolute value
pow(2, 3) # 8; raises a number to the given power
round(12.4567, 2) # 12.46; rounds to given # sig digs
```

A Special Function

There's another built-in function that works differently from the others. `input(msg)` displays a message in the interpreter, lets the user type a response in the interpreter, then stores the response as a string when the user presses enter.

```
input("Enter your name: ") # whatever the user typed
```

This will make it possible for you to write **interactive** programs more easily! This will also let the user **enter data** interactively.

Type Functions

There are a few other built-in functions that are helpful to know, as they let you change the type of data values. This is called **type-casting**, and it is especially useful when you need to change the type of user input.

```
int("4") # 4; converts a value to an integer
float(3) # 3.0; converts a value to a float
str(98.9) # "98.9"; converts a value to a string
bool(0) # False; converts a value to a Boolean

type(4 + 3.0) # float; returns the type of the eventual value
# uses the names we covered before – int, float, str, bool
```

Components of Functions

The functions we call have two core components:

Argument(s) – the values provided inside the parentheses, the **input**

Returned Value – what the function evaluates to after running, the **output**

Arguments Provide the Input

The specific inputs we provide to a function are called **arguments**. These are like the specific bread, peanut butter, and jelly we used in the PB&J algorithm. In the function call `abs(4)`, the argument is `4`.

Arguments are separated by commas and placed between the parentheses of the function call. Functions can require as many (or as few) arguments as needed.

The **positions** of the arguments usually have meaning. In `pow(2, 3)`, the first argument is the base and the second argument is the exponent. In other words, `pow(2, 3)` and `pow(3, 2)` mean two different things.

Receive Output as Returned Value

When a built-in function takes its arguments and runs through its algorithm, we cannot see what it is doing.

When the function is done, it sends back an output as a **returned value**. We usually say a function **returns** a value. This value substitutes in for the function call the same way a variable's value substitutes in for the variable.

For example, the returned value of `pow(2, 3)` is `8`.

Function Calls Follow Order of Operations

Function calls evaluate to a single returned value; that means they are **expressions**. Therefore, we can **nest** function calls inside other expressions the same way we nest basic values and operations.

```
round(pow(abs(-12), 1/2), 2)
```

Just like in math, functions follow order of operations using parentheses. Start by evaluating the inner-most expressions, `abs(-12)` and `1/2`. Then evaluate the call to `pow`; finally, evaluate the call to `round`.

Activity: Write Code Using Functions

You do: write a line of code in the interpreter that takes a variable `x` which holds a number as a string, turns it into an integer, and then doubles that integer.

For example, if `x = "21"`, then your line of code should produce `42`

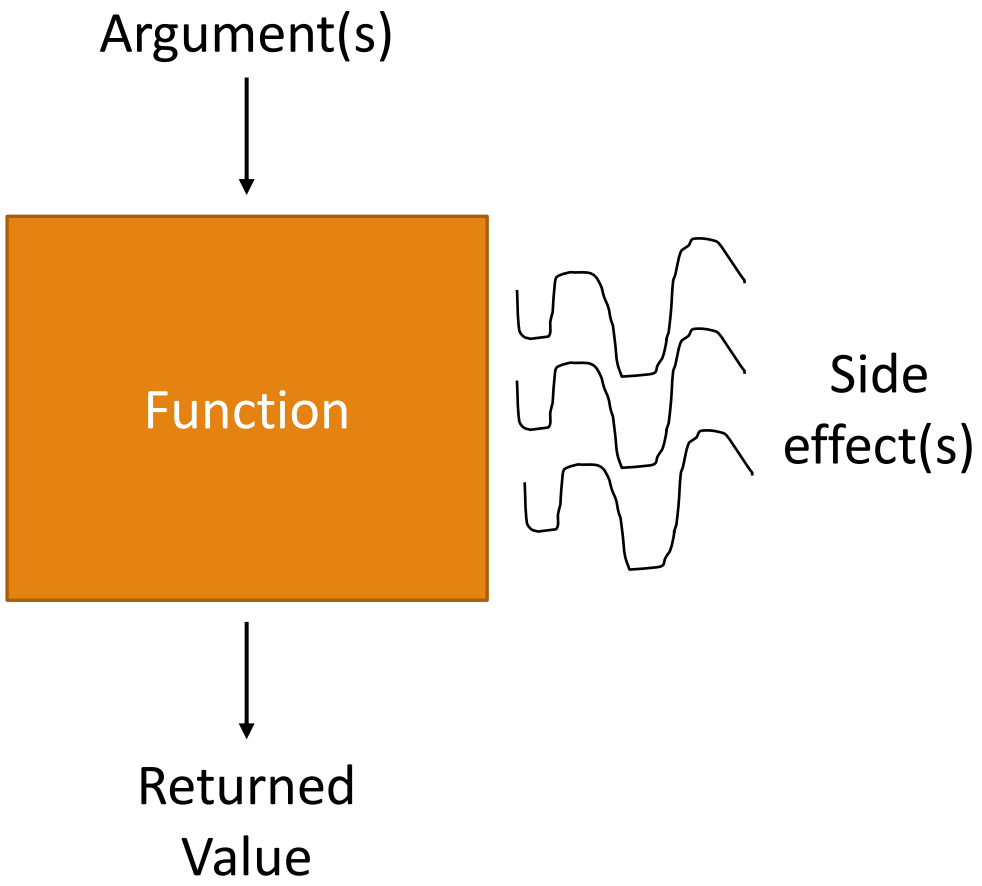
Side Effects Show Change

Recall that a program has a **state** that holds the current information that the program knows (what has been printed, what values do the variables hold).

Function calls themselves are expressions, as they evaluate to a data value (the returned value). But sometimes a function changes the program state in an observable way as it is running; for example, it might display values in the interpreter, or modify a file, or produce graphics. This is called a **side effect**.

If we call `pow(2, 3)`, there is no observable side effect. However, `input("How are you?")` has an observable side effect: it prints a message to the screen and pauses the program until the user responds. `input` also has a returned value – the message typed by the user.

Function Call Process



`print` works differently

Let's take a moment to talk about a function that works in a particularly confusing way: `print`.

`print` takes any number of arguments (the values between the parentheses). It produces a side effect when it concatenates those values and displays the result to the interpreter.

What is `print`'s returned value? It could be the displayed value, but that would let us do weird things like:

```
x = print(2) + 2 # sets x = 4 # but not really!
```

We probably don't want that. Instead, we'll say that `print` has **no explicit output**. But it's not that simple!

```
>>> None
>>> print(None)
 None
>>>
```

Missing Returned Values are **None**

If a function produces no explicit output, it still has a returned value – we need *something* to store in a variable or display. That value is the built-in value `None`.

`None` means that there was no explicit output to be returned. Like `True` and `False`, its meaning is built into Python, so it does not need quotes.

If you try to set a variable to the returned value of a `print` call, you'll find that the variable holds `None`; **print always returns `None`**.

Note that `None` does not show up in the interpreter unless you explicitly `print` it; the interpreter just shows a blank instead.

Activity: Identify the Function Call Parts

Consider the following two function calls. For each function call, what are its **argument(s)** and its **returned value**? Does it have any observable **side effect(s)?**

```
round(3.14159, 1)
```

```
print("15", "-", "110")
```

Libraries

Import Adds Code from Libraries

The Python language has a ton of pre-built functions, but most aren't included in the built-in package (the one available by default). Most of the functions are organized into separate **libraries**.

To use a function from a library, you must **import** the library. This makes it possible to access the functions and variables in that collection. You can do this with the code:

```
import libraryName
```

Library Documentation Organizes Functions

How can you determine which functions exist in which libraries? Read the **documentation**!

All the Python libraries have documentation online that describes which functions are available and what they do. Find it by going to docs.python.org/3/ .

There are a great many libraries and functions, so it's better to check the documentation as needed than to try to memorize all the functions that exist.

Importing the math Library

For example, we can import the **math** library to add more mathematical capabilities. Note that we must put `math.` in front of each function or variable name we use, to specify it came from that library.

```
import math
math.ceil(6.5) # 7; ceiling of a float number
math.log(64, 2) # 6.0; finds the log of 64 with base 2
math.radians(90) # 1.570...; converts degrees to radians
math.pi # 3.141..; it's π!
```

Importing the random library

Importing libraries lets us get more creative with programming. For example, the **random** library lets us generate random numbers, which can help produce novel behavior.

```
import random
random.randint(1, 10) # picks a random int between 1-10 inclusive
random.random() # picks a random float between 0-1
```

Activity: Try Out Libraries

You do: try importing the math or random library in the interpreter and calling functions in them. Read the documentation to see if you can find new functions that they implement. See what you can observe about how they work.

Graphics Library

Importing a graphics library

By using libraries, we can write code that does more than just produce text on the screen. We can even produce graphics with programming! We'll do this with the **tkinter** library, which makes it possible to draw shapes on a separate screen.

```
import tkinter
```

Tkinter Starter Code

We need to run some code before and after our graphics code to make it work.

The `root` is the window. The `canvas` is the thing on the window where we can draw shapes.

The `root.mainloop()` line will tell the window to stay open until we press the X button.

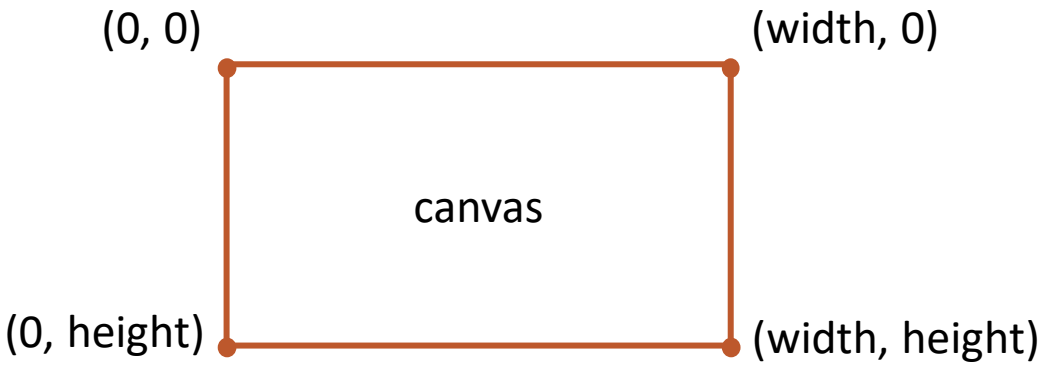
You do: Try copying this code into your editor and running it. You should see a window pop up!

```
import tkinter

root = tkinter.Tk()
canvas = tkinter.Canvas(root,
                        height=400,
                        width=400)
canvas.configure(bd=0, highlightthickness=0)
canvas.pack()

# write your code here

root.mainloop()
```

Coordinates on the Canvas Grow Down-Right

The **canvas** created by the starter code is the thing we'll draw graphics on. It's a two-dimensional grid of pixels. This grid has a pre-set **width** and **height;** the number of pixels from left to right and the number of pixels from top to bottom.

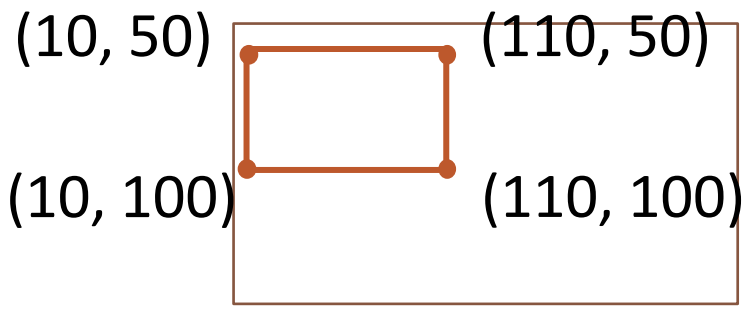
We can refer to pixels on the canvas by their (x, y) coordinates. However, these coordinates are different from coordinates on mathematical graphs – the origin starts at the **top left corner** of the canvas.



Drawing a Rectangle

To draw a rectangle, use the function `canvas.create_rectangle`. This function takes four required arguments: the x and y coordinates of the **left-top** corner, and the x and y coordinates of the **right-bottom** corner. The rectangle will then be drawn between those two points.

```
canvas.create_rectangle(10, 50, 110, 100)
```



Graphics – Side Effects and Returned Values

When the rectangle is drawn on the canvas, we can't use it in future computations. That's a **side effect**.

The graphics function call also returns something – an integer ID associated with the drawn shape. We won't use that value in this class.

Keyword Arguments Add Variety

With the basic parameters, we can only draw outlines of shapes. By adding **keyword arguments**, we can change the properties of these shapes.

A keyword argument is an argument that is associated with a specific name instead of a position in the function call. We can put keyword arguments in any order we like as long as they occur after the main arguments.

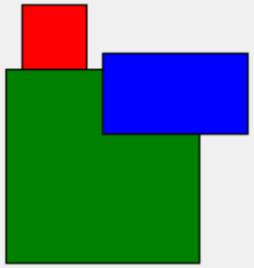
Keyword arguments can have **default values**, which is why we don't need to include them in every graphics call. To change that default value, include the keyword, followed by =, followed by the new value in the function call.

```
canvas.create_rectangle(50, 100, 150, 200, fill="green")
```

Keyword Argument - fill

The `fill` argument can be used on any shape. It uses a string (the name of the color) to change the color of the shape.

```
canvas.create_rectangle(40, 40, 80, 140, fill="red")

canvas.create_rectangle(30, 80, 30 + 120, 80 + 120,
                        fill="green")

canvas.create_rectangle(90, 70, 180, 120, fill="blue")
```



Note that when we draw shapes on top of each other, the one on top is the **last one called**. Order matters!

Interested in finding more Tkinter color names? There's a whole databank!

https://wiki.tcl-lang.org/page/Color+Names%2C+running%2C+all+screens

Activity: Draw Some Rectangles

Try using `canvas.create_rectangle` and the `fill` keyword argument to draw some rectangles on the Tkinter canvas.

Can you get one of the rectangles to draw as a square?

Can you center one of the rectangles?

Problem Solving with Graphics

We can put a rectangle anywhere on the screen by choosing the right arguments, but how do we determine what those arguments should be?

Use **mathematical logic**! Determine where you want the rectangle to be *based on* other locations (the sides of the window or other shapes).

You can use **variables** to give meaningful names to numbers, which may make it easier to represent what you want.

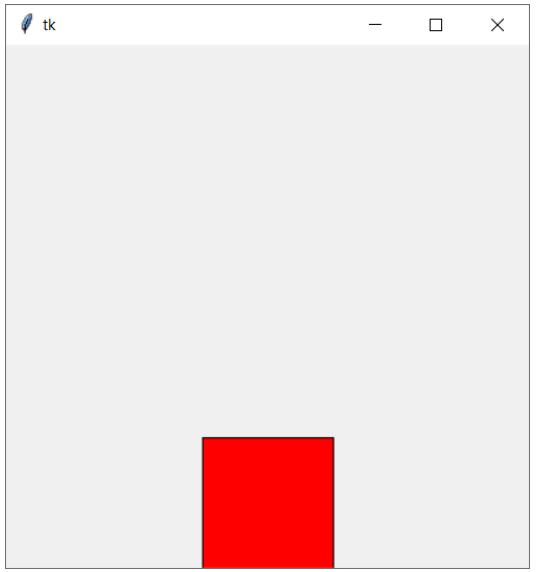
Example: Centering a Square

Problem: we want to draw a square at the bottom-center of the screen, so the bottom edge is touching the bottom of the screen.

What do we know?

- The shape is a **square**, so the height and width must be the same. In other words, (right-left) and (bottom-top) must be the same. Let's call the height and width **size**.
- The shape is **centered horizontally**, so the middle of the square must be in the middle of the screen: screenWidth/2. From the middle, subtract size/2 to get the left position, and add size/2 to get the right position.
- The shape is **aligned with the bottom**, so the bottom must be the same as the screen height. That means the top must be screenHeight - size

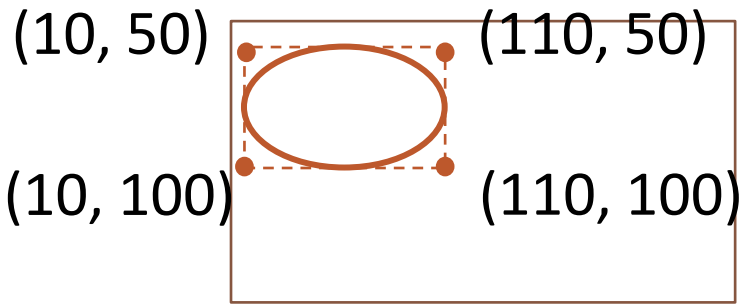
```
w = 400 # screen width
h = 400 # screen height
size = 100 # square side length
canvas.create_rectangle(w/2 - size/2,
                        h - size,
                        w/2 + size/2,
                        h,
                        fill="red")
```



Drawing an Oval

We can draw more shapes than just rectangles. To draw an oval, use `create_oval`. This function uses the same parameters as `create_rectangle`, where the coordinates mark the oval's **bounding box**.

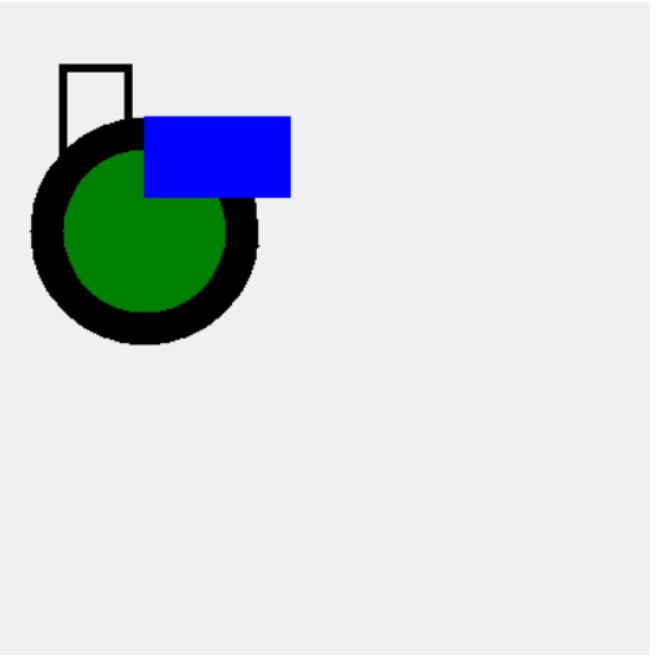
```
canvas.create_oval(10, 50, 110, 100)
```



Keyword Argument - width

Another keyword argument is `width`, which specifies how many pixels wide the border of the shape should be.

```
canvas.create_rectangle(40, 40, 80, 140, width=5)

canvas.create_oval(30, 80, 150, 200,
                   width=20, fill="green")

canvas.create_rectangle(90, 70, 180, 120,
                        fill="blue", width=0)
```

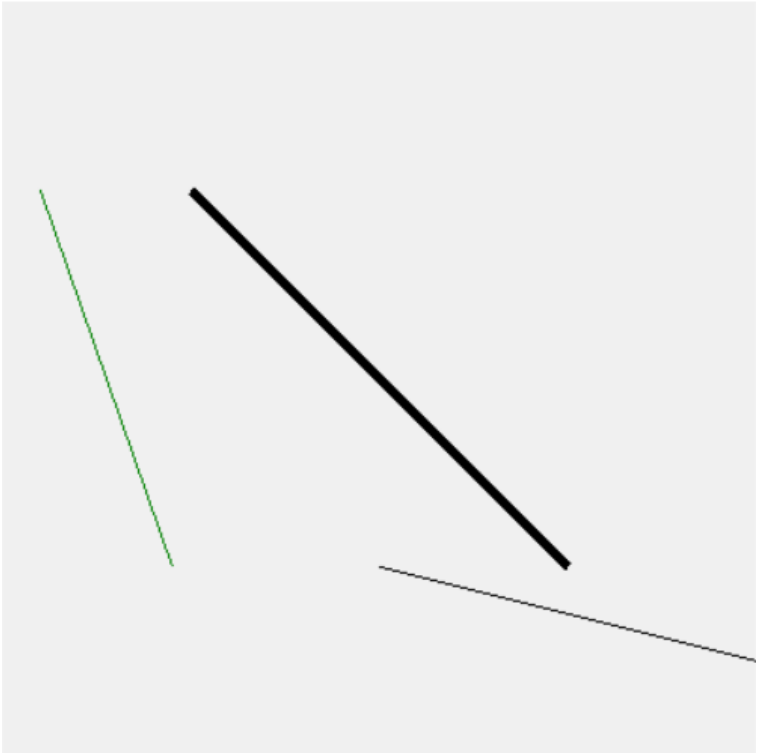


Note that setting `width` to `0` removes the border completely.

Drawing Lines

To draw a line on the screen, you specify the two endpoints of the line.

```
canvas.create_line(200, 300, 400, 350)

canvas.create_line(20, 100, 90, 300, fill="green")

canvas.create_line(100, 100, 300, 300, width=5)
```



Again, we can use `fill` and `width` to modify the lines.

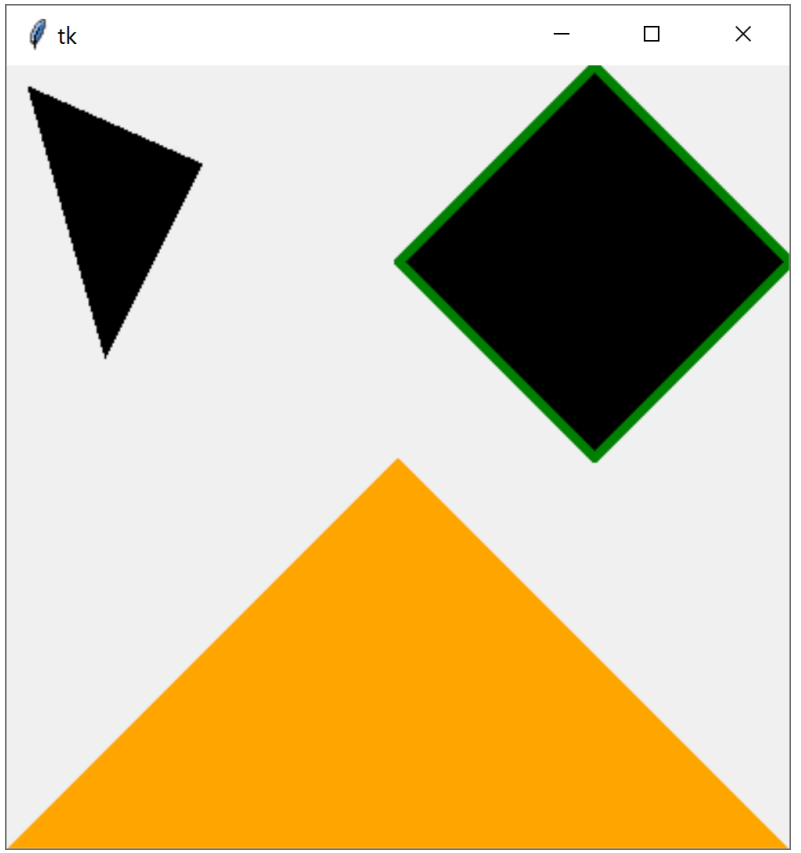
Drawing Polygons

To draw a polygon, you need to specify the coordinates of **each of the polygon's points as an x, y coordinate, in perimeter order**.

The polygon can have as many points as needed but will need at least three points to appear.

```
canvas.create_polygon(10, 10, 50, 150, 100, 50)
canvas.create_polygon(200, 200, 400, 400, 0, 400,
                      fill="orange")
canvas.create_polygon(200, 100, 300, 0, 400, 100, 300, 200,
                      outline="green", width=5)
```

Note here that we've also added a new keyword argument – **outline**, which specifies the color of the shape's outline.



Drawing Text

Drawing text on the canvas works a bit differently from drawing rectangles, ovals, lines, and polygons. We specify only one coordinate – the pixel where the **center** of text will be drawn.

```
canvas.create_text(200, 200, text="Hello World")
```

Although `text` is keyword argument and technically optional, `text` is required in order to draw text at all.

Keyword Argument - font

When drawing text, we can use the keyword argument `font` to change the appearance of the text.

The font parameters takes a string with one to three pieces of information – the font name, the font size, and the font type.

```
canvas.create_text(200, 200, text="Hello World!",
                   font="Arial")

canvas.create_text(100, 100, text="This is fun!",
                   font="Times 30")

canvas.create_text(300, 300, text="weewooweewoo",
                   font="Courier 10 italic")
```

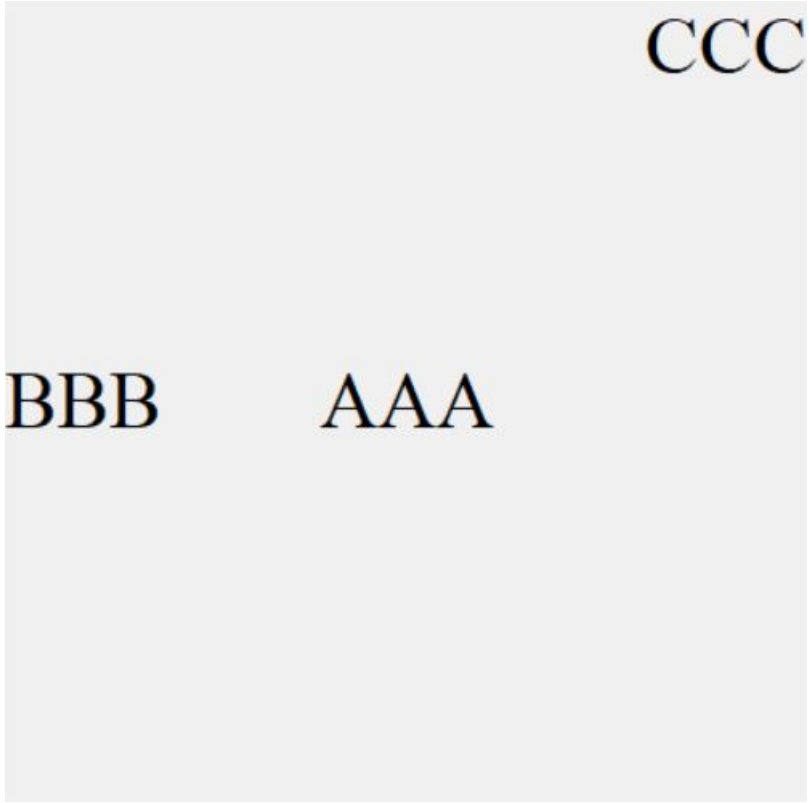
You can find a full list of fonts and types here:
https://effbot.org/tkinterbook/tkinter-widget-styling.htm#fonts

Keyword Argument - anchor

The point used in the `canvas.create_text` call is actually an **anchor** for the text, to describe where it is drawn from. That anchor defaults to the center of the text box, but we can change it to be any compass point instead.

```
canvas.create_text(200, 200, text="AAA",
                   font="Times 30", anchor="center")

canvas.create_text(0, 200, text="BBB",
                   font="Times 30", anchor="w")

canvas.create_text(400, 0, text="CCC",
                   font="Times 30", anchor="ne")
```

Note that the anchor describes the **point on the text box** that will correspond to the (x, y) coordinate. Since CCC's anchor is `"ne"` (north-east), the upper-right corner of the text box is placed at (400, 0).



Drawing images

If we want to use a pre-made image in Tkinter, we can load one in as a PhotoImage. This can be created with:

```
img = tkinter.PhotoImage(file="sample.gif")
```

We can resize the image if needed, using **subsample** to make it smaller and **zoom** to make it bigger.

```
img = img.subsample(5) # make the image 5 times smaller
img = img.zoom(2) # make the image twice as large
```

Unfortunately, PhotoImages can only be .pgm, .ppm, and .gif files. For more filetypes, use the external module Pillow, described in the advanced slides this week.

Drawing images

Once you've created an image, you can draw it with `create_image`. This method takes the x, y coordinates of the image and can have other optional parameters:

```
# the image to be displayed. not really optional...
canvas.create_image(200, 100, image=imageVar)

# the anchor point of the coordinate.
# Same as for text, default "center"
canvas.create_image(200, 100, image=imageVar, anchor="n")
```

Tkinter Can Do Even More!

There's plenty of things Tkinter can draw and plenty of additional keyword arguments that we haven't covered here.

If you're interested in learning more, check out the Tkinter documentation:

https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/index.html

Learning Goals

Identify the **argument(s)** and **returned value** of a function call

Use **libraries** to import functions in categories like math and randomness

Use the **graphics library** to construct images algorithmically