FUNDAMENTALS IN PYTHON & PENNDRAW ANIMATION

LOGISTICS

- HW01 Checkpoint was due last night
  - You can submit these late by using a late token, but we'd strongly suggest not using late tokens on that
- I owe you candy

LAST WEEK WE TALKED ABOUT...

- Variable Assignment
- Conditionals
- Loops

TODAY...

- Transferring these concepts to Python syntax
- Animation in Penndraw
- A new type of loop (while loops)

RECAP AND SYNTAX: VARIABLE ASSIGNMENT

Previously:

```
spreadable = mayo
stackable = cheese
place_bread()
spread(spreadable)
place(stackable)
place_bread()
```

🖨️👇

Similar concept, real Python Syntax

```
spreadable = "mayo"
stackable = "cheese"
print("place bread")
print("spread " + spreadable)
print("add " + stackable)
print("place bread")
```

🖨️👇

```
place bread
spread cheese
add cheese
place bread
```

RECAP AND SYNTAX: CONDITIONALS

Previously:

```
# A program for leaving the house
grab phone

look outside
if it_is_raining:
    grab an umbrella
elif it_is_snowing:
    grab my boots
else: # not raining NOR snowing
    grab a book

walk outside
```

Similar concept, real Python Syntax

```
weather = 2 # 0 = rain, 1 = snow
print("grab your phone")

if weather == 0:
    print("grab an umbrella")
elif weather == 1:
    print("grab boots")
else:
    print("grab a book")
```

SYNTAX CONCEPT: = VS. ==

Python, like many programming languages, uses `=` as its assignment operator

That means that if you want to *assign* something (save it into a variable), you use equals to mark that.

```
course = 1100 # means the variable course now contains the value 1100
```

But what about when we actually want to check if two things are equal??

We use the `==` as our equality operator

```
course == 1100 # gives True or False based on the value of `course`
```

RECAP AND SYNTAX: FOR LOOPS

Previously:

```
aggregator = 1
for i in range(1, 6):
    aggregator = aggregator * i
print(aggregator)
```

Actually that's just regular Python there's nothing to compare here 😄

SYNTAX CONCEPT: INDENTATION AND :

You'll notice a pattern in the format of if statements and for loops

```
for i in range(3):
    print("hello")
```

```
if weather == 0:
    print("grab an umbrella")
```

The `:` tells the computer that the following indented lines are *contained inside* of the structure

Indentation tells the computer what parts of the code should be considered part of the body of the structure

ACTIVITY: FIND THE MISTAKES [C1]

The code snippet on the right is meant to produce the outcome on the left. Find all the errors in the code, write the line numbers and a description of the mistake in [C1]

```
1 x == "cat"
2
3 for i in range(x):
4   for j in range(i):
5   print(x[j])
6
7 print(done)
```

```
c
c
a
c
a
t
done
```

ACTIVITY: FIND THE MISTAKES [C1]

```
1 x == "cat"                <--
2
3 for i in range(x):        <--
4   for j in range(i):      <--
5   print(x[j])             <--
6
7 print(done)               <--
```

COMBINING CONCEPTS

You can use different structures inside of each other.

Remember filtering from last class?

```
for value in sequence:        # For each value in the source sequence,
  if condition(value):        # if that value meets some condition...
    print(value)              # print it!
```

`condition()` is a placeholder here to represent some boolean expression that helps decide whether or not to include `value`.

ACTIVITY: PRINT OUT EVERY VOWEL IN A STRING [C2]

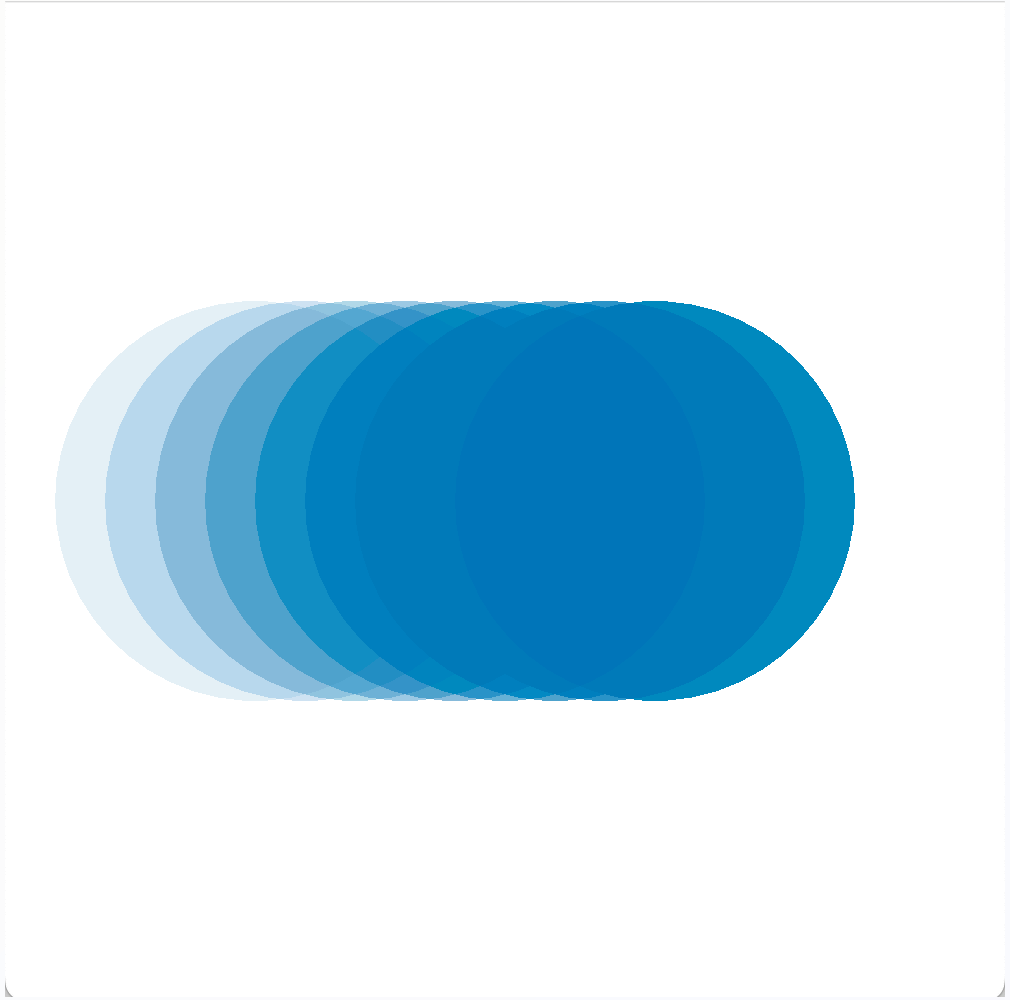
Given a string (e.g., "penguin") print out all of the vowels

Tip: remember that you can check if some string is contained in another string by using `if string_1 in string_2:`

ANIMATION IN PENNDRAW

ANIMATION AS A MODEL

- For static images, we picked the positions, sizes, & colors of our shapes **once**.
- Smooth animation is achieved by showing a lot of similar images very quickly, like a flipbook
  - Screen is updated ~30 times per second
  - Shapes will change position, size, color slightly with each update



FRAMES

- Animations as "flipbooks" require that we draw many images per second
  - Call these images **frames**
  - More frames per second *("higher FPS")* ➡️ smoother animation
- A frame consists of a set of shapes rendered at a specific moment in time
  - Different frames typically have the same shapes but drawn in different ways

Animations usually include a *setup* and an *animation loop.*

```
import penndraw as pd
# SETUP: This code is run just one time!
pd.set_canvas_size(500, 500)
x_center = 0.5
y_center = 0.5
half_side = 0.1
pd.set_pen_color(pd.HSS_BLUE)
# ANIMATION LOOP: This code is run many times per second,
# over and over and over and over again.
while x_center < 1:
    pd.clear()
    pd.filled_square(x_center, y_center, half_side)
    x_center += 0.01
    if x_center > 1 + half_side:
        x_center = -half_side
```

ANIMATING: THE SETUP

Animated programs usually start with a "setup" block, where we:

- choose settings, like canvas size
- declare variables that we will use to draw each frame of the animation
  - variables will vary, but we pick their initial values (how the animation starts)
- do anything that needs to happen **only one time.**

```
import penndraw as pd
pd.set_canvas_size(500, 500)
x_center = 0.5
y_center = 0.5
half_side = 0.1
pd.set_pen_color(pd.HSS_BLUE)
```

ANIMATING: THE *WHILE* LOOP

`while`, like `if` is a keyword that allows us to control the flow of a program.

```
while expression:
  do_this()
  do_that()
```

When we reach the `while`, we test its condition. If `True`, we execute the statements in the body. Then, we **test the condition again**.

- Different from a conditional (`if`), where we only test ONCE!
- If the `expression` is literally `True`, we will loop here forever...

ANIMATING: THE *WHILE* LOOP

The body of the `while` loop is our *animation loop*:

- runs many times per second
- runs until the program is manually stopped or until the condition is `false`
- ➡️ allows us to draw many frames per second, doing something slightly different each time.

ANIMATING: ANIMATION LOOP RECIPE

For each frame,

1. decide whether to clear the screen
    1. Clearing the screen ➡️ all previous shapes disappear, only most recent shape is visible
    2. Not clearing ➡️ most recent frame is drawn on top of other frames, which might still be visible
2. draw the next frame based on current properties of the shapes
    1. "properties of the shapes" usually stored in variables
3. update the properties of the shapes for the next frame
4. `pd.advance()` ➡️ make everything show up on screen
    1. Always need this at the end of the loop.

EXAMPLE ANIMATION LOOP: SLIDING SQUARE

Produces a square that slides left-to-right across the canvas.

```
import penndraw as pd
x_center = 0.5 # SETUP
while x < 10:
    pd.clear()                                  # 1. clear the screen
    pd.filled_square(x_center, 0.5, 0.1)  # 2. draw this frame
    x_center += 0.01                            # 3. update shapes for next fr
    pd.advance()                                # 4. pd.advance()
```

EXAMPLE ANIMATION LOOP: SLIDING SQUARE

Modify the program to include print statements, time tracking:

```
import penndraw as pd
x_center = 0.5 # SETUP
loop_num = 1
while x < 10:
    pd.clear()                                  # 1. clear the screen
    pd.filled_square(x_center, 0.5, 0.1)  # 2. draw this frame
    print(f"In Loop #{loop_num}, square is at x={x_center}")
    x_center += 0.01                            # 3. update shapes for next fr
    pd.advance()                                # 4. pd.advance()
```

EXAMPLE ANIMATION LOOP: SLIDING SQUARE

Modify the program to include print statements, time tracking:

```
import penndraw as pd
x_center = 0.5 # SETUP
loop_num = 1
while x < 10:
    pd.clear()                                  # 1. clear the screen
    pd.filled_square(x_center, 0.5, 0.1)  # 2. draw this frame
    print(f"In Loop #{loop_num}, square is at x={x_center}")
    x_center += 0.01                            # 3. update shapes for next fr
    loop_num += 1
    pd.advance()                                # 4. pd.advance()
```

```
In Loop #1, square is at x=0.5
In Loop #2, square is at x=0.51
In Loop #3, square is at x=0.52
...
```

ANIMATION: ADVANCING

You always need a call to `pd.advance()` at the end of your animation loop. Otherwise nothing shows up.

CONTROLLING THE SLIDE (IF TIME)

```
import penndraw as pd
x_center = 0.5 # SETUP
while x < 10:
    pd.clear()                                  # 1. clear the screen
    pd.filled_square(x_center, 0.5, 0.1)        # 2. draw this frame
    x_center += 0.01                            # 3. update shapes for next fr
    pd.advance()                                # 4. pd.advance()
```

After a while, `x_center` will be very big.

- If `x_center` is much bigger than `1.0`, the square won't be visible at all!
- Want to add some logic to make sure the square resets after a while

ACTIVITY (IF TIME)

If we're always drawing a particular square using this line:

```
pd.square(x, 0.5, r)
```

In terms of `x` and `r`,

...write a boolean expression that will be `True` when the square is completely off of the screen to the right. [L1]

...write a boolean expression that will be `True` as soon as any part of the square is off the screen to the right. [L2]

What is the value of `x` that would draw a square which has its *right* edge just barely touching the *left* boundary of the canvas? [L3]

CONTROLLING THE SLIDE (IF TIME)

What does "off the screen" mean?

- Happens when a shape is all the way off the top, bottom, left, or right sides of the screen.
- If the square is always heading to the right, it'll fall off the right side
- Since the square has a `half_length` of `0.1`, the coordinate of its *left side* is always `x_center - 0.1`.
- ➡️ the square is offscreen when `x_center - 0.1 > 1.0`

CONTROLLING THE SLIDE (IF TIME)

Making the square reset to the left once it disappears:

- After we update the square, check if it's offscreen
- If the square is offscreen, move it all the way to the left of the screen
- If the square is not offscreen, don't do anything else extra

```
import penndraw as pd
x_center = 0.5 # SETUP
while x < 10:
    pd.clear()                                  # 1. clear the screen
    pd.filled_square(x_center, 0.5, 0.1)  # 2. draw this frame
    x_center += 0.01                            # 3. update shapes for next fr
    if x_center - 0.1 > 1.0:
        x_center = -0.1
```