

# Intro to Coding

Class 6

# Functions

- What does a function do?
  - A function takes in **parameters** (a bunch of variables), processes them, and then optionally **returns a value**
  - The term for executing a function is a **function call**
- Vending machine analogy
  - The vending machine takes in money and your input, and returns a corresponding product (a drink or bag of chips)
  - Think of the money and your input as **parameters** to the function, and the product as the **return value**
- Sidenote: when you call a function, you pass **arguments**, not **parameters**
  - Ex.
    - `def greet(name, age):` ← 'name' and 'age' are **parameters**
      - `print(f"Hello, my name is {name} and I am {age} years old.")`
    - `greet("Alice", 30)` ← "Alice" and 30 are **arguments**

# Review: Built-in Functions

- We have gone over two input/output functions:
  - `print(input_string)`
    - Prints the **string** argument `input_string` to the console
    - Non-string variables must be cast to string before printing
  - `input(prompt)`
    - It can **optionally** take in a `prompt` variable, which is similar to using `print(prompt)` then calling `input()`
    - Takes in user input from the console, and returns a **string**
    - The type inputted can be modified via **type casting**
      - Ex. `int(input())`

# Creating Your Own Function

Functions always follow this format

```
def function_name(parameters):
```

- **function\_name** can be anything, as long as it doesn't conflict with other variable/function names in the program
- **Parameters** are the variables that are inputted into the function (these are the coins that are inserted into the vending machine)
  - Note 1: you don't actually type the word "parameters" here, this is just placeholder text to show where the parameters should be
  - Note 2: this can be left empty if your function does not require parameters to work
  - Note 3: multiple parameters should be separated by a comma and space

# Return Values

- This is the output of the function
- It is the equivalent of the vending machine giving you something in return for your money (a drink, a snack, etc.)
- This is **not always required** of a function, and you can choose to omit the return statement

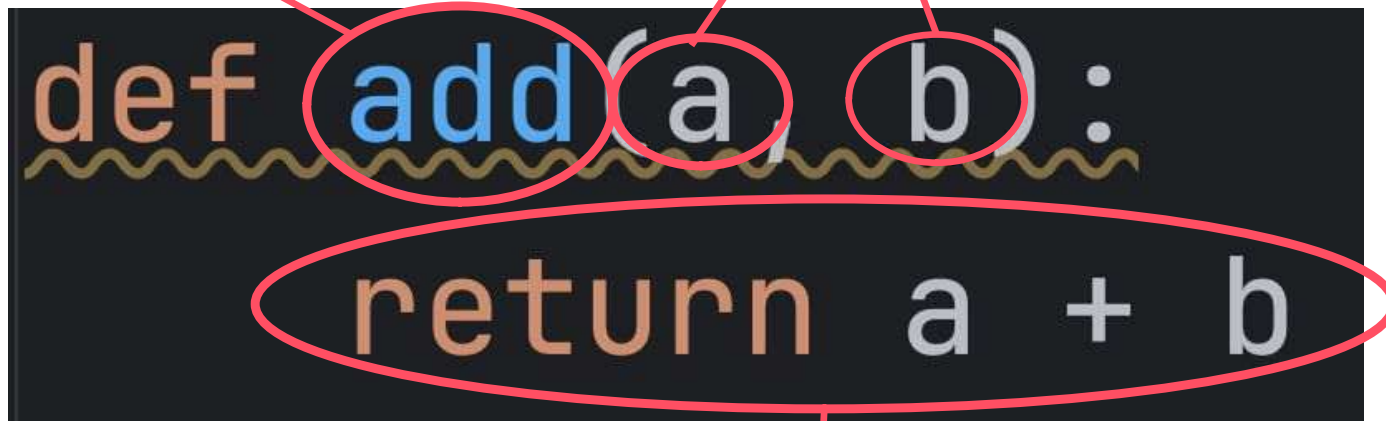
Format: **return** \*value\*

- \*Value\* refers to the variable or literal value being returned
- You can also directly return the result of an equation instead of storing it in a variable
- When a function returns a value, any subsequent lines of code **will not run**

## Sample Function: Number Addition

Function name

Parameters a  
and b



```
def add(a, b):  
    return a + b
```

The image shows a Python code snippet on a dark background. The code is `def add(a, b):` followed by a wavy line and `return a + b`. Annotations include: a red arrow pointing to `add` from the label 'Function name'; a red arrow pointing to `a` from the label 'Parameters a and b'; a red arrow pointing to `b` from the label 'Parameters a and b'; a red oval around the `return a + b` statement with a red arrow pointing to the label 'Return statement which yields the sum of a and b'.

Return statement which  
yields the sum of a and b

## Sample Function: Number Addition

```
def add(a, b):  
    return a + b
```

Any indented lines below the `def` statement are part of the function

```
add(1, 2)  
add(8, 5)
```

Since these lines are not indented, they are not part of the function

## Functions Activity

Using what you have just learned about functions, create a function that **takes in two legs lengths** of a triangle, then **solves for the hypotenuse**.

Pythagorean theorem:  $a^2 + b^2 = c^2$  (where a and b are the leg lengths, and c is the hypotenuse length)

**Note:** pythag is the custom function's name

```
print(pythag(a: 5, b: 7))
```

```
8.602325267042627
```

```
print(pythag(a: 3, b: 4))
```

```
5.0
```



## Solution

- Import the `math` library to access the `sqrt` function
- Since  $c^2 = a^2 + b^2$ , the function will return the square root of  $a^2 + b^2$ , which is equivalent to  $c$

```
import math

def pythag(a, b):
    return math.sqrt(a ** 2 + b ** 2)

print(pythag(5, 7))
print(pythag(3, 4))
```

# Classes & Objects

- A class is like a **blueprint** for creating objects.
- In games (which we will create using Pygame), we use classes to build **reusable components with predictable behavior**, like:
  - Players
  - Enemies
  - Bullets
  - Obstacles
- Using classes allows us to:
  - Organize code neatly
  - **Reuse** code easily
  - **Group data and actions together**
- Objects are the actual instantiations of these blueprints
- Methods are functions inside of classes

# Classes & Objects

```
import pygame
```

```
class Player:
```

```
    def __init__(self, x, y, size, color):
```

```
        self.x = x
```

```
        self.y = y
```

```
        self.size = size
```

```
        self.color = color
```

```
        self.speed = 5
```

```
    def draw(self, screen):
```

```
        pygame.draw.rect(screen, self.color, (self.x, self.y, self.size, self.size))
```

The first and **REQUIRED** method is `__init__()`. It allows us to **create objects** and outlines all the data that the objects will **store** in variables, which are called **attributes**.

# Classes & Objects

- Create a Player object like this: `object = Class(attributes)`
  - The `__init__()` method runs when an object is created from the class even without actually calling “`__init__()`”:

Object →

```
player1 = Player(200, 200, 50, (0, 128, 255))
```

→ Class

- Attributes:
  - The numbers within the highlighted pink box are arguments passed to the `__init__()` function to become the object's attributes
  - They can be **accessed within the class itself** and in the **objects created with the class**

```
def __init__(self, x, y, size, color):  
    self.x = x  
    self.y = y  
    self.size = size  
    self.color = color  
    self.speed = 5
```

We are making data that the object **stores**.  
This data can be read and modified

In this case, the x attribute stores the  
x-coordinate of the object

# Self Parameter

- `self` parameter

- `self` refers to the **specific object** that is being created or used.
- When we make a class (ie. `Player`), we will create many different objects from it
- Each `Player` object will have **its own** position, color, speed, etc.
- To refer to one specific object within the class, we can use the **parameter** `self`

```
def __init__(self, x, y, size, color):  
    self.x = x  
    self.y = y  
    self.size = size  
    self.color = color  
    self.speed = 5
```

Run a method **for the specific object it is called for**.  
We will talk more about methods shortly!

Assign/access the **attribute of the individual object**

## Classes & Object Activity

Create a class `Player`, which stores information about the player. It should include the following attributes:

- Name
- Current Health
- Maximum Health
- Attack Power
- Inventory (represented by a list of strings)

*Note: do not delete your code, as you will need to expand upon it in the next activity!*

## Solution

```
class Player:
    def __init__(self, name, health, max_health, attack, inventory):
        self.name = name
        self.health = health
        self.max_health = max_health
        self.attack = attack
        self.inventory = inventory

p = Player("CodeUp", 100, 100, 5, [])
```

# Methods

```
import pygame

class Player:
    def __init__(self, x, y, size, color):
        self.x = x
        self.y = y
        self.size = size
        self.color = color
        self.speed = 5

    def draw(self, screen):
        pygame.draw.rect(screen, self.color, (self.x, self.y, self.size, self.size))
```

All objects of this class can use its methods. Remember classes **group data** (attributes) **and actions** (methods)



# Methods

- Methods are functions inside of classes
- Calling a function:
  - Format: `object.method(parameters)`

```
def draw(self, screen):  
    pygame.draw.rect(screen, self.color, (self.x, self.y, self.size, self.size))
```

```
player1.draw(screen)
```

When we call any method **on player1**, the **self argument** is automatically passed to the method (we don't need to write it again!). This allows for the methods in a class to **act independently on one object**.

# Overall Example

Sample code for a class named "Cat".

```
1 class Cat:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
5     def meow(self):
6         print("Meow! My name is " + self.name + " and my age is " + str(self.age))
7
8 my_cat = Cat("Tom", 7)
9 my_cat.meow()
10
```

self must always be the first parameter. It is **implicitly passed** into the function. Then, you can add the other parameters that need to be **explicitly passed into the function**.

Assigns values to the object's attributes

When an object is created, the program will attempt to call the `__init__` function. Passes "Tom" as name and 7 as age. Python will automatically recognize `my_cat` (object name) as the `self` parameter for both `__init__()` and `meow()`

Meow! My name is Tom and my age is 7

## Class Functions Activity

Using your code from the PLayer class activity, add the following functions:

`damage(d)` - the player has received `d` damage, they will lose `d` health points. **The player should not go below 0 health.**

`consume(h, a)` - the player consumes an item that recovers `h` health and increases their attack power by `a`. **The player should not go above their maximum health, but there is no limit to their attack power.**

`collect(item)` - the player collects an item, adding it to their inventory. Assume the player has an infinite inventory size.

# Solution


We can use the `min` and `max` functions to limit the player's health

```
class Player:
    def __init__(self, name, health, max_health, attack, inventory):
        self.name = name
        self.health = health
        self.max_health = max_health
        self.attack = attack
        self.inventory = inventory


    def damage(self, d):
        self.health = max(self.health - d, 0)

    def consume(self, h, a):
        self.health = min(self.health + h, self.max_health)
        self.attack += a

    def collect(self, item):
        self.inventory.append(item)
```



```
p = Player("CodeUp", 100, 100, 5, [])
p.damage(5)
print(p.health)
p.collect("Sword")
p.collect("Bow")
print(p.inventory)
p.consume(100, 5)
print(p.health)
print(p.attack)
```



```
95
['Sword', 'Bow']
100
10
```

# Pygame

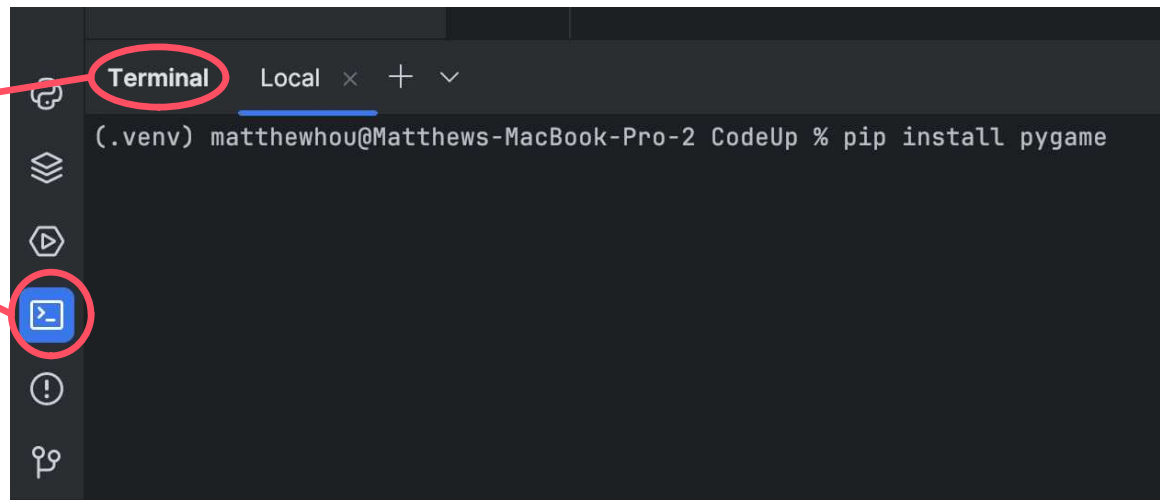
Pygame is a module for python designed for writing simple video games. Steps to starting and stopping pygame properly:

1. Install the module
2. Import it into code with `import pygame`
3. Call `pygame.init()` to start Pygame
4. Import the sys library with `import sys` to properly close the program when you are done

# Installing Pygame

In Pycharm, open a new Terminal (located in the bottom left-corner of your screen) and type “pip install pygame” and then press enter. Pygame should automatically install. This process is the same for both Mac and Windows.

Ensure that it says Terminal and that the correct icon is chosen



# Initial code for Pygame

Some code is needed at the start of nearly every pygame project:

```
resolution = (800, 400)

screen = pygame.display.set_mode(resolution)

pygame.display.flip()
```

This creates a 800 x 400 window. This is also known as the **surface variable**. Make sure it is declared **outside the while loop**. (refer to the next slide)

```
pygame.display.set_caption('CodeUp Project')
```

This names the window “CodeUp Project”. Note that a caption is optional.

There are some other variables we need to create; these will be looked at later.

# Setting up the Pygame Loop

To keep the window open until user quits, we can write this:

```
while True:  
    for event in pygame.event.get():  
        if event.type == pygame.QUIT:  
            pygame.quit()  
            sys.exit()
```

Constantly checks for new events

Because of the while True loop, the game will run indefinitely until the user decides to close the window.



# Your First Pygame Program

Copy and paste the code on the Google Classroom, then run it!

```
import sys
import pygame

resolution = (800, 450)
color = (235, 0, 0)
screen = pygame.display.set_mode(resolution)

while True:
    screen.fill(color)
    pygame.display.flip()

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
```

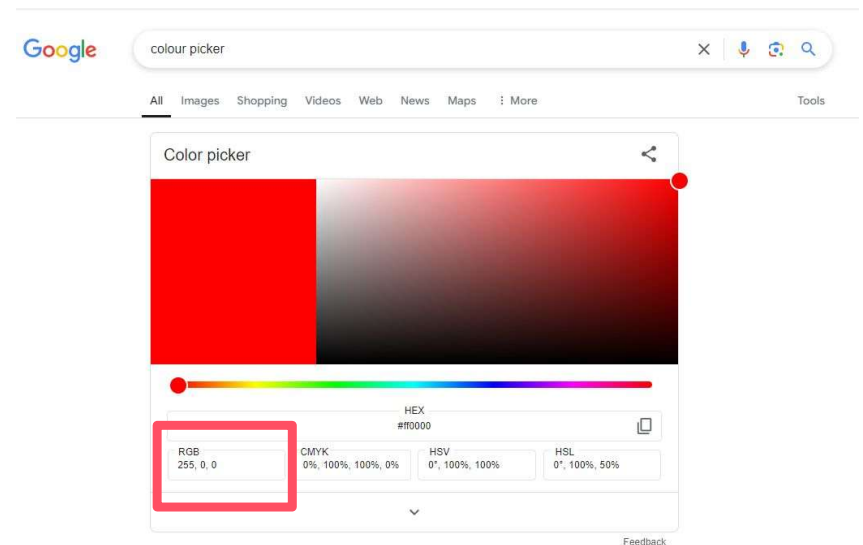
# Your First Pygame Program



# RGB Codes

- Colours are displayed in the form (R, G, B)
  - R = red value, G = green value, B = blue value
- The value for each variable is at least 0 and at most 255 inclusive
  - The higher the number, the more vibrant/intense the colour and vice versa

Search up “colour picker” on Google



# Homework

Try playing around with some of the variables you see. How might you change the colour? How might you change the window size? Try using different RGB values for the “color” variable. What do you notice?