Helpful Script Notes

func \_physics\_process(delta):

* This tells my game to check something 60x per second (e.g., to see if the player is pushing an arrow key)
* This might be useful for data collection purposes later?
* Source: approx. 23 minutes <https://youtu.be/WpzI2ytz5MA>

var PlayerName = “[username inserted here]”

* I have not quite figured out how to assign usernames to players/participants but I assume these types of variables will be needed
* Source: approx. 27 minutes <https://youtu.be/WpzI2ytz5MA>

If then statements/Boolean Expressions

* E.g., if Input.
  + A variety of actions (e.g., is.action.pressed)
  + Source: approx. 35 minutes <https://youtu.be/WpzI2ytz5MA>

Changing the background/lighting of the game (for the different fair/equal/control levels)

* Approx. 10 - 19 min in Part 4 <https://youtu.be/j2wBsb-W3Ls> how to create different sky colours (again for different levels)

Maybe use a ‘for loop’ for determining how many times a player needs to play a certain level (e.g., in the forced choice trials?)

* <https://youtu.be/GS-lAZBNm6k>

Can also potentially benefit from some matching statements for figuring out when to run certain levels vs. others

* <https://youtu.be/q20R9rAM6Z4>

Use arrays for creating a scoreboard?

* <https://youtu.be/HCHWhy-Mru4>
* \*\*Example in the functions video about PlayerHealth at approx. 9 minutes, this might be what I need for the adding the player’s score to the scoreboard?? <https://youtu.be/Ze1mK6I-9QA>
* Exporting arrays <https://youtu.be/_oquCodoVl8>

Random Number Generator Coding

* Brief mention in this video <https://youtu.be/ykMssEpDPfc>

\*\*All the class tutorials in this series (<https://youtu.be/ykMssEpDPfc>) might be useful for creating a class of ‘mock players’ to get the random scores and such for the scoreboards?

\*\* Part 8 around 50 minutes for getting the game to go to the scoreboard screen <https://youtu.be/rS-PcTTiGtA>

\*\*Part 3 around 28.5 minutes is how you can assign different keys to different functions (e.g., making w mean go up like the up arrow) <https://youtu.be/VWy4ErYd630>

\*\*Rewatch Part 3 tutorial around 11 minutes to see how to change import files to scenes to use them as props in game possibly? <https://youtu.be/VWy4ErYd630>

**Thoughts:**

* Should I try to make there be a certain number of cans per square area?
* I could have the scoreboard automatically appear after 30 seconds in each level rather than having a separate scoreboard scene? Might make linking everything for flow easier?

**GDSCript Notes**

* If stuck on something:
  + Google
  + Stack Overflow/GodotEngine Questions
  + Ask questions in the YouTube comments of the videos
* Data Types
  + Tells the compiler how the programmer intends to use the data
  + Define the operations that can be done on the data, the meaning of the data, and the way values of that type can be stored
* Literal Value (AKA “Literal”)
  + A value written directly in the source code as opposed to being the result of some other expression (e.g., variable or constant)
  + You ‘literally’ provide them in your script/code
* Common data types:
  + Strings (e.g., “Hello world!”)
    - Notice the quotations – these are important, tells the compiler it’s a string value
    - Used to represent text instead of numbers
    - Comprised of a set of characters including spaces and numbers
      * Examples of a string literal: “I have 0 cats”, “Hello world!”, “123”
    - Cannot use strings for math equations/operations
    - String numbers do not have the same value as integer values
      * i.e., “1” + “1” does not equal 1 + 1 (without quotations, so using integers)
        + “1” + “1” will equal 11, because it will compile the two separate string literals into one string
        + 1 + 1 will equal 2, because the integers make a math operation
  + Numbers/Integers (e.g., 0, -12, 200)
    - Can be negative, positive, or zero
    - No decimals – only whole numbers
    - Represents some range of mathematical whole numbers
      * In Godot, the integer range is large - huge number, don’t need to know
        + If you go outside of the range, you will overflow and wrap around, and it will just go to the range extent instead of the actual number outside of the range
  + Floats (e.g., 0.00, -12.00, 200.00, 12.32)
    - Similar to integers
    - Have decimals
    - A floating-point number – a number that has a decimal place
    - Used when more precision is needed
  + Booleans (e.g., true, false; else if, if statements, etc.)
    - Two values only – true and false
    - True and false values are used to represent two truth values of logic and Boolean algebra
    - In GDScript, when converted to numeric values, false = 0 (or 0.0) and true = 1 (or 1.0)
  + Null (e.g., null)
    - Only null value
    - Used to represent the absence of data
    - It’s an empty data type
      * Contains no information
      * Cannot be assigned any other value
* Variable (AKA “Scalar”)
  + A storage address paired with an associated symbolic name, which contains a known/unknown quantity of information (i.e., a value)
    - A container that contains a value
  + How to write a variable
    - Type out ‘var’ followed by a unique name
      * A variable cannot have a name that already exists in your script
    - E.g., var uniqueName ( = Literal Value/Data Object)
      * Anything in the brackets is optional
        + You can make the variable empty (will contain data type ‘null’ by default)
        + Or, you can assign the variable a literal value or data object using the equal symbol

E.g., var playerHealth = 100

E.g., var playerHealth

* + Typed Variables
    - Data type that is explicitly declared on the variable by adding the colon symbol and the data type in which you wish to limit your variable
      * i.e., the variable called lifePoints can only have integer values assigned to it
      * E.g., var lifePoints: int = 100
      * E.g., var lifePoints: int
    - Declaring data types on variables is usually good practice
      * Leaves less room for confusion on the purpose of the specific variable
    - Assigning a value that is not the declared type will throw an error
      * E.g., var text: String = 10
        + i.e., 10 is not a string, but an integer
    - Values assigned to a typed variable must have a compatible type
      * Can coerce a value to be a certain type
        + Use the casting operator ‘as’ (i.e., will convert a value to the desired value)

E.g., var text: String = 10 as String

Will turn 10 to “10”

E.g., var num: int = “10” as int

Will turn “10” to 10

E.g., var numFloat: float = 10 as float

Will turn 10 to 10.00

* + - Common data types:
      * var aNumber: **int** = 10
      * var aBoolean: **Boolean** = true
      * var aString: **String** = “Hi friend”
      * var aFloat: **float** = -12.00
    - Can also infer data types by replacing the above bolded words with := and ensuring the literal values are included
      * var aNumber := 10
      * var aBoolean := true
      * var aString := “Hi friend”
      * var aFloat := -12.00
      * := means inferring (i.e., ‘:’) and assigning (i.e., ‘=’)
    - Can also assign an object that is the subtype of the cast type (advanced topic – different video)
      * E.g., var my\_node2D: Node2D = $Sprite as Node2D
        + Sprite is a subtype of Node2D
        + Casting Sprite as a Node2D
  + Actual format for declaring variables:
    - var uniqueName ( : Data Type) ( = Literal/Data Object)
      * See all the examples above
      * In brackets are optional
  + When to use variables
    - When you have data that needs to change over the lifetime of the game/program
      * E.g., var playerHealth = 100
        + The # value of health may change
  + Memory Life Cycle
    - The time between an object’s creation and it’s destruction
    - The same in all programming languages
      * Allocate memory 🡪 Use memory 🡪 Release memory
      * Allocate Memory
        + Memory is allocated by the operating system

This allows your program to use it

* + - * + All computers have memory (i.e., RAM)
        + Memory can be thought of as an array of bytes
      * Use Memory
        + After memory is allocated, it can then be used
        + Read/write operations are happening as you are using the allocated memory

Such as assigning and changing variable values in your script/code

* + - * Release Memory
        + When variable(s) are no longer in use, the operating system will release the entire memory so that it is freed and can be used to allocate new memory
        + This is done manually or automatically, depending on the programming language
        + The operating system only determines how much memory is allocated to the application, but the application’s programming language determines how memory is handled

The less memory you use, the better – better performance across computers and operating systems

* + - Basic Variable Process
      * E.g., var x = 2020
      * Step 1. Allocate enough memory for the variable, in this case enough memory to hold an integer
      * Step 2. Assign the value 2020 to that memory address location
      * Step 3. Indicate that x points to that value
      * Step 4. Depending on the language, memory is freed when not in use
      * For more info on this: 5min <https://youtu.be/9LaB6wbZepg>
        + Godot uses reference counting
      * Nodes in Godot are not referenced counted in the engine
        + Removing a node from the tree will not delete it (even ‘remove\_child’ does not remove it from memory)
        + To better manage nodes, create a pool of objects and re-use the node object
        + Can explicitly destroy/free the node from memory by calling ‘queue\_free()’ in the ‘\_exit\_tree()’ which is called when the node is removed from the tree
* Operands
  + Numerical, text, and Boolean values that a program can manipulate
  + Can be object values
  + E.g., in 1 + 2 the 1 and 2 are Operands
  + E.g., in x – y the x and y are Operands
* Operators
  + Symbols used to manipulate and check operand values
  + E.g., in 1 + 2 the + is an Operator
  + E.g., in x – y the – is an Operator
  + Assignment Operators
    - Assign a value to its left operand based on the value of its right operand
    - The simplest assignment operator is the ‘=’ which assigns the value of its right operand to the left operand (e.g., x = y; i.e., we are assigning the variable x with the value stored in the variable y)
    - Examples of assignment operators
      * The operand on the left will be assigned (i.e., ‘=’) the values on the right
      * x = x + y, **x += y** (additional assignment)
      * x = x – y, **x -= y** (subtraction assignment)
      * x = x \* y, **x \*= y** (multiplication assignment)
      * x = x / y, **x /= y** (division assignment)
      * x = x % y, **x %= y** (remainder assignment)
        + Gives you the remainder of a division
      * The bolded versions are the shortcut versions
  + Comparison Operators
    - Compares its operands and returns a logical value based on whether the comparison is true
      * The logical value will be a Boolean value
    - To see if two operands are equal, use == (the equal operator; e.g., x == 1)
      * == is an equal operator whereas = means to assign a value
    - To see if two operands are not equal, use != (the not equal operator; e.g., x != 1)
      * What you are trying to convey is that x does not equal 1 and if x is a value of 1, you’ll get ‘false’ but if it is any other number than 1 then you will get ‘true’
    - To see if the left operand is greater than the right operand, use > (the greater than operator; e.g., x > 1)
      * If x is 2 or more, it will come back as ‘true’ but if x is 1 or 0 or any less then it will come back as ‘false’
    - To see if the right operand is greater than the left operand, use < (the greater than operator pointing the other direction; e.g., x < 1)
    - For greater than or equal to, use >= or <=
  + Logical Operators
    - 3 of them:
      * && (Logical AND operator)
        + Can be denoted by ‘&&’ or ‘and’
        + It will return false as long as any of the operands can be converted to false, starting from the left most operand
        + Example:

var x = true

x && 1

We are checking to see that x and 1 can be converted to true

The logical operand will first check to see if the left operand can be converted to false

If it can, the value false will be returned

But, in the above example x = true

If instead the example had x = false, then it would return false

So, it will check to see if the second operand can be converted to false

If it can, the value false will be returned

In the above example 1 = true (i.e., 1 is true, 0 is false)

So, in this example it will return true

* + - * || (Logical OR operator)
        + Can be denoted as ‘||’ or ‘or’
        + Return true as long as any of the operands can be converted to true (starting from the left most operand)
        + Example:

var x = false

x || 1

We are checking to see if x or 1 can be converted to true (checks x first, then 1)

In this example, x = false, so the operator will move on to check the 1, which can be converted to true

Because 1 can be converted to true, the logical OR operator will return ‘true’

If in the above example we changed x to = true, the logical OR operator would see that x can be converted to true, and wouldn’t need to check the 1 before returning ‘true’

The difference between AND and OR is that AND will check both operands, whereas OR will only check the operands until one can be converted to true (it stops operations once it hits an operand that can convert to true)

* + - * ! (logical NOT operator)
        + Denoted by ‘!’ or ‘not’
        + Can be used to check a single operand
        + It will return false if its operand can be converted to true, otherwise it will return true
        + Example:

var x = false

!x

We are checking to see if x can be converted to True

In this case, since the operand cannot be converted to true (because x = false), then the not operator will return true

But, if the x = true, then the operator would return false

For not operators, whatever value can be returned as the variable, the operator will return the opposite (i.e., if x is true, then not will be false, if x is false, then not will be true)

If the variable is false, the !operator will always be true

* Constants
  + A value that never changes
  + E.g., const ValueNeverChanges = 1
  + E.g., const playerSpeed = 5
  + E.g., cons Speed = 5
  + Constants, like variables, need to have a unique name
  + You cannot assign an empty constant
    - A constant needs to be assigned a value
  + Like variables, can also add an explicit data type
    - E.g., const ValueNeverChanges: int = 1
    - This is not usually necessary though because the data type is inferred with the assigned value (and the value never changes, so there’s likely to be no confusion)
  + Constants are read only
    - Modifications are not permitted after declaration and assignment
    - E.g.,
      * const ValueNeverChanges = 1
      * After this is assigned, if you then add something to the script like ValueNeverChanges = 100, an error will be thrown, because 100 does not equal 1
  + Pros for using constants (instead of assigning literals):
    - Readable code
    - Easier debugging
    - Saves time when editing
    - Constants and literals are the same, and you can use literals instead of constants
      * But it makes the code easier to read if you use constants, and it’s easier for debugging
      * E.g., 3 min in <https://youtu.be/q-qmjicdM3k>
  + Use constants when you want to declare a value that you know will never change in the lifecycle of your game/program
    - E.g., const PlayerHeight = 7
  + Naming conventions when using constants
    - Make sure the naming conventions for constants are visually different from the variables
      * E.g., const PlayerName = “John” (all constants starting with an uppercase; or could do all uppercase for constants e.g., PLAYERNAME)
      * E.g., var playerHealth: int = 100 (all variables starting with a lowercase)
* Comments
  + Programmer reader annotations in source code
  + Use # to denote a single line comment
    - Anything written after the # is considered a comment
    - Won’t be a part of the code
    - Anything before the # will be considered code, everything after the # will be skipped by the code
  + Use “”” at the beginning and the end of a multi-line comment
    - It is read as a multi-line string by the compiler
  + Types of comments:
    - Methodology description
      * Used to explain methodology used in your code
      * Used to explain the code rather than a clarification of the code’s intent
      * E.g., a comment to explain why insertion sort was used instead of quick short
    - Meta-data
      * At the top of the script
      * Can include company name, file name, year script was created, people who maintain the script, copyright, etc.
    - Debugging
      * Most likely using a brute force debugging method
      * E.g., commenting print statements
        + Usually comment these print statements out afterwards, with the intent to use them again in later debugging
    - Code description
      * Used to make others understand the intent of the line of code
      * Different than explaining the methodology
      * Used only when needed
  + Use comments to tell you why, let the code tell you how
  + Do not use comments to
    - Source/version control – to explain who did what and what people changed and when
* Naming Conventions
  + A set of rules for choosing the character sequence to be used for identifiers which denote variables, types, functions, and other entities in source code and documentation
  + Using bigger names is okay, computers can handle it
  + The intention of the variable/function should be clear from its name
    - The code can explain the intent and purpose of the variables/functions
    - Do not use comments to explain your name/purpose
      * If you need to write a comment to understand the purpose, rethink the name
  + Avoid using data types in the name (e.g., intHealth 🡪 don’t do this, just do 🡪 Health: int = value)
  + Use searchable names
  + Decide a standard for names
    - Makes it consistent across the board
    - Uppercase for Constants, lowercase or camelCase for variables and functions
* If/Elif/Else Statements (https://youtu.be/huxEefIqfIc)
  + These are control statements
    - Control the flow of execution
  + A type of control statement is the conditional statement
    - If, else if, and else are conditional statements
  + if/elif/else are control structures that execute a block of statements if a certain condition is true and a second block of statements if the condition is false
  + If Statements (can be single or multiple)
    - An if statements starts the if else statement chain
    - Only one if statement is allowed to be used per chain
    - An If statement checks a test
      * If the test is true, then it executes everything in the statement block
      * If the test is false, then it skips everything in the statement block
    - E.g.,:
      * if <test>:
        + <statement>
        + \*\*if it’s true, the statement is run, if it’s false, it gets skipped and moves to the next if statement or exits the chain to the next line of code
      * if <test>:
        + <statement>
  + if/elif combo
    - The else-if has to precede the if statement
    - You can provide multiple else if statements in the chain
    - E.g.,:
      * if <test>:
        + <statement>
        + \*if it’s true here, this statement is followed through; if it’s false it moves on to the elif test and statement
      * elif <test>:
        + <statement>
        + \*if it’s true here, it runs the statement, if it’s false, it skips the statement and exits the chain
  + if/else statement
    - Else statements must be at the end of the chain, and it ends the if/elif/else chain
    - Else executes a statement if all tests fail in the chain
    - E.g.,:
      * if <test>:
        + <statement>
        + \*if it’s true here, the statement is followed through; if it’s false, it goes to the else statement
      * else:
        + <statement>
        + \*The else block gets executed if everything before is false
  + if/elif/else statement chain (AKA “Nested If/Else Statement”) example:
    - if <test>
      * <statement>
    - elif <test>
      * <statement>
      * \*might have multiple elif statements
    - else:
      * <statement(s)>
  + The chronological order is important!
* While Loop
  + A control flow statement
  + Allows code to be executed repeatedly based on a given Boolean condition
  + Code:
    - while <test>:
      * <statement>
      * \*if the test is true, the statement is run, if the test is false we exit the while loop
  + As long as the test remains true, the loop and statement will be run repeatedly (creates an infinite or endless loop)
    - As soon as it is false, the loop is exited
  + Infinite/Endless Loops
    - A piece of coding that lacks a functional exit so that it repeats indefinitely
    - The opposite = a loop
      * A sequence of instructions that is continually repeated until a certain condition is reached
    - 3 types infinite loops:
      * Fake
        + They give the impression that it is an infinite loop
        + But it actually isn’t an infinite loop
        + E.g.,

while (true):

break

\*the while(true) implies an infinite loop, but the second line is a break line which means the loop will only run once before exiting the chain

* + - * Intended
        + The user intentionally doesn’t want the loop to end
        + When the game runs, you are running an intended infinite loop – the game wants to run everything indefinitely until the player wants to exit the game
      * Unintended
        + Should be avoided
        + The user/coder made a mistake and forgot to exit out of the loop by accident
        + This will freeze the app/game
* For Loop
  + A control flow statement
  + Allows code to be executed repeatedly
  + Godot uses iterator based for loops
    - Allows for the enumeration of sets of items other than number sequences
  + A for loop is used when you want to iterate through a range, such as numbers, arrays, tables
    - The range is what determines how many times we’re going to run the code inside the block statement
    - When looping through an array, the current element is stored in the loop variable
    - When iterating through a dictionary, the index is stored in the loop variable
  + Code:
    - for <placeholder> in <range>:
      * <block statement>
    - E.g.,:
      * for x in 10:
        + <block statement>
        + \*\*10 is the range, therefore the block will execute 10 times

When using numbers, the variable in the for loop starts at 0, and increments until the variable is greater than or equal to the number used as the range will be run

* + Example Code for a for loop with an array:
    - for x in arrayitem:
      * <block statement>
      * \*will execute for as many items as in the array
* Match Statements
  + Type of selection control mechanism
  + Allows the value of a variable or expression to change the control flow of the program execution via search and map
    - Basically, you use a variable to ‘match’ a pattern and execute the code inside the match block statement
  + Patterns are matched in chronological order
    - If a pattern matches, the matched block will be executed, then the statement will exit the chain
  + Can use an ‘\_’ at the bottom of a match statement to denote a default block statement to run
    - If you would like to continue after matching, use the ‘continue’ keyword
  + Match Statement Code Example:
    - match x:
      * 1:
        + print(“Hi friend”)
        + \*\*if you add continue here, the match block will continue checking other patterns instead of exiting the chain
      * \_:
        + print(“\_is a wildcard”)
      * \*\*the x value needs to match one of these patterns (i.e., 1 or \_) in order to run the statement (i.e., the print statement)
  + 7 types of match patterns
    - Constant pattern
      * Uses constant primitive data types as your match pattern
      * E.g. Booleans, strings, integers, etc.
      * Can use literal values
    - Variable pattern
      * Consists of variables and enums
    - Wildcard pattern
      * Can use the ‘\_’ symbol for the wildcard pattern
        + Used to match everything
        + Used as a default block
      * The wildcard/default pattern is best to use last in the chain
    - Binding pattern
      * Catches everything (like the wildcard statement), BUT it assigns the match value into a variable
      * You declare a variable into the code, which can be included into the block statement
    - Array pattern
      * Every single element in the array is a pattern in itself
      * Arrays can be nested
      * You can use sub-patterns inside array patterns
    - Dictionary pattern
      * Works like the array pattern, but instead of arrays, using dictionaries
    - Multiple pattern
      * Can specify multiple patterns as long as they are separated by commas
      * Cannot have bindings to them
  + Coding example: <https://youtu.be/q20R9rAM6Z4>
    - Matching can help test big if statement chains
      * Helps to make code cleaner than massive if statement chains
* Arrays
  + A collection of items stored at contiguous memory locations
    - When stored in memory address, it’s done sequentially
  + Used to store multiple items together
  + Can contain elements of different data types, arrays, dictionaries
  + An array is like a line, there’s someone at the front (position 0), someone at the back, and some in between
  + A value in an array = an element
  + The position in an array = Index Position (starts at 0)
  + Can use inverse index positions to start at the back of the array (e.g., -1, -2, -3, for the last, second to last, etc. position in the array)
  + Declaring an empty array:
    - var emptyArray = []
  + Declaring an array with data (predetermined values):
    - var filledArray = [1,”hi”,true]
      * \*can have different data types, but must be separated by commas
  + Retrieving data from an array:
    - Just type out the variable name
    - E.g., filledArray would get you the entire array
    - E.g., filledArray[0] would get you the element in the first position of the array
  + Sub-Arrays
    - Creating a sub-array
      * var filledArray = [[1,2],[3,4]]
    - Retrieving data from a sub-array
      * filledArray[0][1]
        + \*first element (sub-array), and the specific value within that sub-array
  + Push and Pop Array Methods
    - Push adds an element to either the beginning or the end of an array
      * E.g., arrays.pop\_back()
        + Removes the last element in the array
      * E.g., arrays.pop\_front()
        + Removes the first element in the array
    - Pop removes and returns an element from the beginning or end of an array
      * E.g., arrays.push\_back()
        + Adds an element to the back of the array
      * E.g., arrays.push\_front()
        + Adds an element to the front of the array
  + Clear an Array
    - Three ways to clear an array
      * arrays = []
        + Reassigning to a clear array
      * arrays.resize(0)
        + Can shorten an array too with this
      * arrays.clear()
        + This is the most intuitive
  + Duplicate an Array
    - var duplicateArray = arrays.duplicate(true)
      * Returns a copy of your array
    - Deep vs. Shallow duplication/copy
      * Deep copy = all nested arrays and dictionaries are duplicated and will not be shared with the original array
      * Shallow copy = references to the original nested arrays and dictionaries are kept, so that modifying a sub-array or dictionary in the copy will also impact those referenced in the source array
      * Can do a shallow copy by changing true to false
  + Get the length of an array
    - var length = arrays.size()
  + Arrays and Dynamic Memory
    - Arrays are objects stored in memory
    - <https://youtu.be/Ty4O74t7koM>
* Strings
  + Any value contained in “”
    - A literal string value
  + How are strings stored in memory?
    - <https://youtu.be/Cmk9DidN6tA>
  + Strings in For Loop
    - You can loop your string variable/values
  + Strings have index positions – if you want to only pull out portions of your string, you can do that
  + There are a lot of string methods
    - E.g., insert 🡪 insert values into an index position of an array
    - E.g., length 🡪 returns the value of the length of the string
    - E.g., split 🡪 turns the string into an array based on the delimiter value
* Enums
  + A data type that contains a fixed set of constants
  + Short for enumeration
  + Good at assigning consecutive integers
    - The integers are always assigned starting at 0 (i.e., 0, 1, 2, 3, etc.)
  + Declaring an Enum
    - enum { left, right, front, back }
      * \*\*put the names of your constants in the brackets
      * \*this would be the same as writing:
        + const left = 0
        + const right = 1
        + const front = 2
        + const back = 3
    - Can change the assigned values:
      * enum { left, right, front = 10, back)
        + \*Would be the same as:

const left = 0

const right = 1

const front = 10

const back = 11

* + - * \*Can assign any integer to any enum constant
  + Can also create an Enum Set (so the constants will not be global, they’ll only come out when the set is called)
    - E.g., enum MoveSet { left, right, front, back }
      * \*Then calling on MoveSet or things like MoveSet.left later
  + Examples he used in the code were Enums like Neutral, Evil, Good, etc. with constant values
    - <https://youtu.be/xaEd02zP7D8>
* Dictionaries
  + Associative container that contains values referenced by unique keys
  + AKA a key-value store
  + Format:
    - var <name> = { literal value : value }
      * \* literal value is the key, and the value is the key value
    - Examples:
      * var emptyDictionary = {}
      * var dictionaryContainer = {“name” : “John”}
      * var anotherContainer = {1: “John”}
      * var complexContainer = {1: [1,2,3]}
  + Accessing Dictionary
    - E.g., dictionaryContainer[“name”]
      * Will return “John”
    - E.g., anotherContainer[1]
      * Will return “John”
  + Adding Keys and Clearing a Dictionary
    - <https://youtu.be/Y1o0ev6VqXs>
  + Dictionary comparisons
    - You cannot do direct comparisons with ==
    - But can if you use the hash method
* Functions (<https://youtu.be/Ze1mK6I-9QA>)
  + A function is a block of organized, reusable code that is used to perform a single, related action
  + Called other things in other programming languages
  + Always part of a class
  + Can return back data (though this is optional)
  + Scope priority
    - Local variables 🡪 class members 🡪 global variables
    - i.e., variables defined at the function level can only be used at the function level
      * AKA Function Scope
  + Functions Format
    - func <name>(<parameter>):
      * pass
        + \*the pass keyword does nothing except to prevent compiler errors from empty functions

Empty functions throw errors

* + - * + \*pass can also be used in loops to avoid errors
  + Creating a basic function
    - func nameOfFunction():
      * pass
      * \*can have empty brackets or parameters in the brackets
  + Return Statements (see video)
    - Void keyword (see video) 🡪 don’t want to return anything
  + \*\*Example in the video about PlayerHealth at approx. 9 minutes, this might be what I need for the scoreboard??
    - <https://youtu.be/Ze1mK6I-9QA>
* Scope Levels
  + Scope = the region of a computer program where name binding is valid
  + Name binding = An association of a name to entity, such as a variable
  + Scopes can vary in range
    - Small loops to entire applications
  + Levels of Scopes
    - Global Scope
      * A class, variable/value, or function in the global scope that can be used anywhere in the entire program
      * E.g., Extends Node # Node is globally accessible
    - File/Module/Class Scope
      * A variable/value or function that can only be accesses from within the file
      * E.g., Your class is the .gd file
    - Function Scopes
      * A variable/value that is only visible inside the function it is declared in
      * E.g., Parameters and variable declarations
    - Code Block Scopes
      * A variable/value that is only visible inside code blocks
      * E.g., variables declared in if statements, for loops, etc.
  + Scope Hierarchy
    - Global (Top) 🡪 File/Module 🡪 Function 🡪 Code Block (Bottom)
    - They go top down, not bottom up
* Classes
  + Describes the contents of the objects that belong to it
  + Classes describe an aggregate of data fields, such as variables, and defines the operations, such as methods
  + Think of a class as a blueprint for creating objects, with initial value states, and implementation behavior
  + By default, all script files are unnamed classes
    - This means you can only reference them by using the file’s path name: using absolute or relative path
      * E.g., res://
      * E.g., C:\
      * E.g., ./FilePath
  + Register Scripts as a Class
    - If you want to give a name to a class use the class\_name keyword followed by a unique name
      * This will register your class name as a new type in Godot’s editor
        + E.g., class\_name Player
      * After your register the Class, you can use the new registered name in other scripts
        + E.g., extends Player

Inherits the class

* + - * + E.g., var a = Player.new()

Creates an instanced object

* + Instancing a class
    - Use new() class function
  + Basic Class Format
    - Step 1 – Inherit the Node 🡪 this is important if using in a scene that extends the node
      * Can be a script or a global class
    - Can have member variables
    - Then have functions (scope hierarchy)
  + Inheriting
    - Inheritance is a mechanism where you can derive a class from another class for a hierarchy of classes that share a set of attributes and methods
    - Use extends keyword
      * Let’s you use Godot’s Global Classes and their methods
      * E.g., extends <filePath or Global Name of class>
    - A class can inherit the following
      * Global Class
      * Another Class File
      * An inner class inside another class file
    - Multiple inheritance is not allowed
  + Virtual Methods (e.g., func \_ready())
    - A method that can be redefined in derived classes
    - Has an implementation in a base class as well as the derived class
    - Used when a method’s basic functionality is the same but sometimes more functionality is needed in the derived class
* Objects
  + An instance of a class
  + AKA class instance, instanced object, class object
  + Instantiation
    - AKA construction
    - The realization of a predefined object
    - An instance of an object may be declared, given a unique name, so that it may be used in a program
  + Instance Variable
    - A member variable in which each instantiated object of the class has a separate copy, or instance
      * This means that data changes to variables in one instanced object will not change in another
    - If you declare a member variable with a fixed data type, the compiler will do its best to type cast to the specified data type if you accidentally try to pass in a different data type
  + When to use an instanced object
    - When you want multiple items/copies of the class (e.g., multiple enemies)
    - Example: Enemy.gd
      * class\_name Enemy (\*you are registering the name to the editor)
      * var a = 100
  + Create Class Object
    - Go to your scene script
    - Create the variables
      * var objectOne = Enemy.new() (\*you use the class name you registered)
      * OR:
      * var loadClass = load(<filepath for the enemy script file>)
      * var objectTwo = loadClass.new()
  + Using a Class Object
    - Create the variable
      * var object = Enemy.new()
    - Then use it
      * func\_start()
        + object.printA()
  + How to make objects Unique
    - Boring to have multiple objects with the same values
    - Power of instances is variation in object properties (e.g., Member Variables)
    - To make objects unique from other objects, simply use the built-in class constructor
      * \_init() method
      * E.g., func \_init(parameter)
        + pass
* Class Inheritance
  + Inheritance is a mechanism where you can derive a class from another class for a hierarchy of classes that share a set of attributes and methods
  + Can inherit a global class, another class file, and an inner class in another class file
  + Use extends keyword
    - Use class name or file path
  + Subclass
    - The class that inherits properties from another class, which is referred to as the super class/derived class
    - Super class
      * The class whose properties are inherited by a sub class/derived class
  + Properties
    - A broad concept, encompasses attributes such as class members and methods, and relationships to other classes such as inherited classes
  + Passing values (i.e., \_init() function)
  + Inheritance allows for cleaner code
    - Makes it easier for us to define a class, especially when we want to make additional tweaks to a class
    - Basically, you use inheritance when you want to reuse code from a class
      * E.g., you might have code for animals in general (Animal.gd) and then you might inherit that code for other animal codes to derive from (e.g., Frog.gd, Horse.gd, etc.)
  + Inheritance allows overriding functions
    - Allows a sub class to provide a specific implementation of a method that is already provided by one of its super class
    - But you cannot override variables
  + You can have multiple layers of inheritance
    - E.g., Node > Animal > Horse
    - Think of scope hierarchy
* Classes as Data Types
  + Classes are reference types (i.e., Objects)
  + Objects are created in memory and then handled through a separate reference
  + Can cast using the ‘as’ keyword/function
    - Think of casting hierarchy
* Duck Typing
  + A type of system used in dynamic languages where the type or class of an object is less important than the method it defines
  + You do not check the types, but rather we check for the presence of a given method or attribute
  + ‘is’ keyword
    - A keyword used to check if a value or object is of a certain type
    - Will return a true/false Boolean
* Static Functions
  + A member function of a class that can be called even when an object of the class is not initialized
  + A static class cannot access any variables of its class except for static variables
    - GDScript does not have static variables
  + You don’t need to create a class object in order to use the functions
  + Best to use static functions when you don’t depend on any class members/variables
  + Code:
    - static func print()
* Export Keyword
  + Class members can be exported to the Godot application using the ‘export’ keyword
  + Benefit = easier to edit values of member variables
  + Benefit = good way of testing game items without having to go inside the script and manually changing code
    - E.g., position, size, speed, etc.
  + Exports must have a variable initialized with a value being assigned to it, or a data type declared for it
    - Editor needs to know what kind of display box to show you for editing data
  + Can export Arrays
* Setter and Getter Functions
  + Getters and Setters are methods used to protect your data, particularly used when creating classes
  + Used when:
    - You want to know when a classes member variables have changed
    - You want to perform an action when a specific value has been changed or retrieved
    - You want to encapsulate variable access in some way
  + setget keyword
  + setFunction
    - func setterFun(parameter)
      * pass
      * \*parameter must be declared
  + getFunction
    - func gettingFunc()
      * pass
      * \*parameter must be empty
  + ‘self’ keyword
    - Refers to current class instance
* Fake Protected Variables
  + Private variables
    - Variables that are visible only to the class they belong to
    - Protected from outside access
    - No way to declare private variables in Godot
      * But we can mimic protected variables
* Memory Management
  + By default, all classes that don’t define an inheritance will automatically inherit the Reference Class
    - Your class will use reference counting (see earlier notes on this)
    - Instances will be freed when no longer in use
  + The global reference class is the base class for any object that keeps a reference count
  + Objects are released when the reference count is 0 (i.e., no longer in use)
  + Might need to use the free function to manually free memory
    - If you do not free objects, you may get a memory leak
  + Code:
    - instance.free()
    - \*instance would be the name of your class object that is inheriting from a class that inherits from the object class (top class, scope hierarchy)
  + When using the node class
    - Use queue\_free() method
    - Comes in with built in safety checks when the node class deletes itself from memory
* Cyclic Dependency
  + AKA circular dependencies
  + A relation between two or more objects which either directly or indirectly depend on each other to function properly
  + The last object references the first, resulting in a closed loop
  + May get an error based on circular dependency
  + The problem with them
    - Can crash a program
    - Can create high coupling
    - Can create a “God class”
    - Leads to Spaghetti Code
  + Godot can detect these before running the game and catch it for you
    - There will be an error
  + Fixing them depends on what you’re trying to accomplish
  + Best way to avoid, Decouple your code
    - Make sure scripts don’t point to each other
    - Do you need 2 scripts that reference each other? Or can it be done in 1?
    - Or create a 3rd class (C) that is the mediator between the two related classes (A and B)
* Signals
  + Single Responsibility Principle
    - Every module or class should have responsibility over a single part of the functionality provided by the software/game
  + Observable Pattern
    - Where an object (i.e., the subject) maintains a list of its dependents (i.e., the observers) and notifies them automatically of any state changes, usually by calling one of their methods
    - Example:
      * Player = Subject
      * Health = Observer
      * When the player gets hurt, we want the subject/player to emit a signal to the observer/health to let it know to react (i.e., decrease the health)
    - The subject doesn’t care who the observer is, just that it needs to send a signal
    - The observer doesn’t care who they are connected to (i.e., the subject), just concerned about reacting to the received signal
    - Subjects can emit to multiple Observers
    - Observers can react to multiple signals/Subjects
  + To use signals in classes/scripts, you must have the Object Class as part of its inheritance chain
  + How to declare a signal
    - 1. Declare a signal name on the Subject Class
    - 2. Declare a connection on the Observer Class
    - Use ‘signal’ keyword
    - Use ‘emit\_signal’ keyword
    - \*\*This is I guess a different way to code in signals other than just clicking on the signals in the side panel??\*\* Makes me wonder if some of the things he’s describing to code are easier in Godot and can just be clicked (e.g., like making sub-resources unique)\*\*
* There are other videos but I think he’s coding at a higher level than I need