COMPUTER PROGRAMMING
LEARNING GUIDE 9

Boolean Expressions

Here is a little example of boolean expressions:

```
a = 6
b = 7
c = 42
print(1, a == 6)
print(2, a == 7)
print(3, a == 6 and b == 7)
print(4, a == 7 and b == 7)
print(5, not a == 7 and b == 7)
print(6, a == 7 or b == 7)
print(7, a == 7 or b == 6)
print(8, not (a == 7 and b == 6)
print(9, not a == 7 and b == 6)
```

Type it in and run it…study the output, comparing it to the program. Be sure you understand every line!

What is going on? The program consists of a bunch of funny looking `print` statements. Each print statement prints a number and an expression. The number is to help keep track of which statement you are dealing with. Notice how each expression ends up being either `False` or `True`. In Python `false` can also be written as 0 and `true` as 1.

The lines:

```
print(1, a == 6)
print(2, a == 7)
```

print out a `True` and a `False` respectively just as expected since the first is true and the second is false. The third print, print(3, a == 6 and b == 7), is a little different. The operator `and` means if both the statement before and the statement after are true then the whole expression is true otherwise the whole expression is false. The next line, print(4, a == 7 and b == 7), shows how if part of an `and` expression is false, the whole thing is false. The behavior of `and` can be summarized as follows:

expression	result
true `and` true	true
true `and` false	false
false `and` true	false
false `and` false	false

Notice that if the first expression is false Python does not check the second expression since it knows the whole expression is false. Try running `False and print("Hi")` and compare this to running `True and print("Hi")` The technical term for this is short-circuit evaluation

The next line, `print(5, not a == 7 and b == 7)`, uses the `not` operator. `not` just gives the opposite of the expression. (The expression could be rewritten as print(5, a != 7 and b == 7)). Here is the table:

expression	result
`not` true	false
`not` false	true

The two following lines, `print(6, a == 7 or b == 7)`and `print(7, a == 7 or b == 6)`, use the `or` operator. The `or`operator returns true if the first expression is true, or if the second expression is true or both are true. If neither are true it returns false. Here's the table:

expression	result
true `or` true	true
true `or` false	true
false `or` true	true
false `or` false	false

Notice that if the first expression is true Python doesn't check the second expression since it knows the whole expression is true. This works since oris true if at least one half of the expression is true. The first part is true so the second part could be either false or true, but the whole expression is still true.

The next two lines, `print(8, not (a == 7 and b == 6))`and `print(9, not a == 7 and b == 6)`, show that parentheses can be used to group expressions and force one part to be evaluated first. Notice that the parentheses changed the expression from false to true. This occurred since the parentheses forced the notto apply to the whole expression instead of just the `a == 7`portion.

Here is an example of using a boolean expression to find a duplicate in a list:

```
list = ["Life", "The Universe", "Everything", "Jack", "Jill", "Life", "Jill"]

# make a copy of the list. In a later learning guide you will see what [:] means.
copy = list[:]
# sort the copy
copy.sort() prev = copy[0]
del copy[0]

count = 0

# go through the list searching for a match
while count < len(copy) and copy[count] != prev:
        prev = copy[count]
        count = count + 1

# If a match was not found then count can't be < len
# since the while loop continues while count is < len
# and no match is found

if count < len(copy):
        print("First Match:", prev)
```

And here is the output:

```
First Match: Jill
```

This program works by continuing to check for match `while count < len(copy) and copy[count] is not equal to prev`. When either `count`is greater than the last index of `copy`or a match has been found the `and`is no longer true so the loop exits. The `if`simply checks to make sure that the `while`exited because a match was found.

The other "trick" of `and`is used in this example. If you look at the table for `and`notice that the third entry is "false and false". If `count >= len(copy)`(in other words `count < len(copy)`is false) then `copy[count]` is never looked at. This is because Python knows that if the first is false then they can't both be true. This is known as a short circuit and is useful if the second half of the `and`will cause an error if something is wrong. We used the first expression (`count < len(copy)`) to check and see if `count`was a valid index for `copy`. (If you don't believe me remove the matches "Jill" and "Life", check that it still works and then reverse the order of `count < len(copy) and copy[count] != prev` to `copy[count] != prev and count < len(copy)`.)

Boolean expressions can be used when you need to check two or more different things at once.

A note on Boolean Operators

A common mistake for people new to programming is a misunderstanding of the way that boolean operators works, which stems from the way the python interpreter reads these expressions. For example, after initially learning about "and" and "or" statements, one might assume that the expression `x == ('a' or 'b')` would check to see if the variable `x` was equivalent to one of the strings `'a'` or `'b'`. This is not so. To see what we mean, start an interactive session with the interpreter and enter the following expressions:

```
>>> 'a' == ('a' or 'b')
>>> 'b' == ('a' or 'b')
>>> 'a' == ('a' and 'b')
>>> 'b' == ('a' and 'b')
```

And this will be the unintuitive result:

```
>>> 'a' == ('a' or 'b')
True
>>> 'b' == ('a' or 'b')
False
>>> 'a' == ('a' and 'b')
False
>>> 'b' == ('a' and 'b')
True
```

At this point, the `and` and `or` operators seem to be broken. It doesn't make sense that, for the first two expressions, `'a'` is equivalent to `'a'` or `'b'` while `'b'` is not. Furthermore, it doesn't make any sense that `'b'` is equivalent to `'a'` and `'b'`. After examining what the interpreter does with boolean operators, these results do in fact exactly what you are asking of them, it's just not the same as what you think you are asking.

A COMPUTER DOES WHAT YOU TELL IT TO DO,
WHICH IS NOT ALWAYS WHAT YOU WANT IT TO DO.

When the Python interpreter looks at an `or` expression, it takes the first statement and checks to see if it is true. If the first statement is true, then Python returns that object's value without checking the second statement. This is because for an `or` expression, the whole thing is true if one of the values is true; the program does not need to bother with the second statement. On the other hand, if the first value is evaluated as `false` Python checks the second half and returns that value. That second half determines the truth value of the whole expression since the first half was false. This "laziness" on the part of the interpreter is called "short circuiting" and is a common way of evaluating boolean expressions in many programming languages.

Similarly, for an `and` expression, Python uses a short circuit technique to speed truth value evaluation. If the first statement is `false` then the whole thing must be false, so it returns that value. Otherwise if the first value is true it checks the second and returns that value.

One thing to note at this point is that the boolean expression returns a value indicating `True` or `False`, but that Python considers a number of different things to have a truth value assigned to them. To check the truth value of any given object `x`, you can use the function `bool(x)` to see its truth value. Below is a table with examples of the truth values of various objects:

True	False
True	False
1	0
Numbers other than zero	The string 'None'
Nonempty strings	Empty strings
Nonempty lists	Empty lists
Nonempty dictionaries	Empty dictionaries

Now it is possible to understand the perplexing results we were getting when we tested those boolean expressions before. Let's take a look at what the interpreter "sees" as it goes through that code:

First case:

```
>>> 'a' == ('a' or 'b')          Look at parentheses first, so evaluate expression "('a' or 'b')"
                                 # 'a' is a nonempty string, so the first value is True
# Return that first value: 'a'
>>> 'a' == 'a'                   # the string 'a' is equivalent to the string 'a', so expression is True
True
```

Second case:

```
>>> 'b' == ('a' or 'b')          # Look at parentheses first, so evaluate expression "('a' or 'b')"
                                 # 'a' is a nonempty string, so the first value is True
                                 # Return that first value: 'a'
>>> 'b' == 'a'                   # the string 'b' is not equivalent to the string 'a', so expression is False
False
```

Third case:

```
>>> 'a' == ('a' and 'b')         # Look at parentheses first, so evaluate expression "('a' and 'b')"
                                 # 'a' is a nonempty string, so the first value is True, examine second value
                                 # 'b' is a nonempty string, so second value is True
                                 # Return that second value as result of whole expression: 'b'
>>> 'a' == 'b'                   # the string 'a' is not equivalent to the string 'b', so expression is False
False
```

Fourth case:

```
>>> 'b' == ('a' and 'b')         # Look at parentheses first, so evaluate expression "('a' and 'b')"
                                 # 'a' is a nonempty string, so the first value is True, examine second value
                                 # 'b' is a nonempty string, so second value is True
                                 # Return that second value as result of whole expression: 'b'
>>> 'b' == 'b'                   # the string 'b' is equivalent to the string 'b', so expression is True
True
```

So Python was really doing its job when it gave those apparently bogus results. As mentioned previously, the important thing is to recognize what value your boolean expression will return when it is evaluated, because it isn't always obvious.

Going back to those initial expressions, this is how you would write them out so they behaved in a way that you want:

```
>>> 'a' == 'a' or 'a' == 'b'
True
>>> 'b' == 'a' or 'b' == 'b'
True
>>> 'a' == 'a' and 'a' == 'b'
False
>>> 'b' == 'a' and 'b' == 'b'
False
```

When these comparisons are evaluated they return truth values in terms of True or False, not strings, so we get the proper results.

Examples

password1.py

```
## This program asks a user for a name and a password.
# It then checks them to make sure that the user is allowed in.

name = input("What is your name? ")
password = input("What is the password? ")
if name == "Josh" and password == "Friday":
        print("Welcome Josh")
elif name == "Fred" and password == "Rock":
        print("Welcome Fred")
else:
        print("I don't know you.")
```

Exercises

1. Write a program that has a user guess your name, but they only get 3 chances to do so until the program quits.

2. Write a program that has a user guess what number you enter at runtime, the number is between 0 and 1001. (Hint: run the program, enter the number. Then the program will print a whole bunch of blank lines to scroll the number off the screen before the user is shown the screen and asked to start entering guesses.) After each guess you will tell the user whether their entry to too high or too low.

Dictionaries

Welcome to Python Dictionaries! Dictionaries have keys and values. The keys are used to find the values. Here is an example of a dictionary in use:

```
def print_menu():
        print('1. Print Phone Numbers') print('2. Add a Phone Number') print('3. Remove a Phone
        Number') print('4. Lookup a Phone Number') print('5. Quit')
        print()

numbers = {} menu_choice = 0 print_menu()
        while menu_choice != 5:
                menu_choice = int(input("Type in a number (1-5): "))
        if menu_choice == 1:
                        print("Telephone Numbers:")
                        for x in numbers.keys():
                                print("Name: ", x, "\tNumber:", numbers[x])
                        print()
        elif menu_choice == 2:
                print("Add Name and Number")
                name = input("Name: ")
                phone = input("Number: ") numbers[name] = phone
        elif menu_choice == 3:
                print("Remove Name and Number")
                name = input("Name: ")
                if name in numbers:
                        del numbers[name]
                else:
                        print(name, "was not found")
        elif menu_choice == 4:
                print("Lookup Number")
                name = input("Name: ")
                if name in numbers:
                        print("The number is", numbers[name])
                else:
                        print(name, "was not found")
        elif menu_choice != 5:
                print_menu()
```

This program is similar to the name list earlier in the chapter on lists. Here's how the program works. First the function `print_menu` is defined. `print_menu` just prints a menu that is later used twice in the program. Next comes the funny looking line `numbers = {}`. All that this line does is to tell Python that `numbers` is a `dictionary`. The next few lines just make the menu work. The lines

```
for x in numbers.keys():
    print("Name:", x, "\tNumber:", numbers[x])
```

go through the dictionary and print all the information. The function `numbers.keys()` returns a list that is then used by the `for` loop. The list returned by `keys()` is not in any particular order so if you want it in alphabetic order it must be sorted. Similar to lists the statement `numbers[x]` is used to access a specific member of the dictionary. Of course in this case `x` is a string. Next the line `numbers[name] = phone` adds a name and phone number to the dictionary. If `name` had already been in the dictionary `phone` would replace whatever was there before. Next the lines

```
if name in numbers:
    del numbers[name]
```

see if a `name` is in the dictionary and remove it if it is. The operator `name in numbers` returns `true` if `name` is in `numbers` but otherwise returns `false`. The line `del numbers[name]` removes the key `name` and the value associated with that key. The lines

```
if name in numbers:
    print("The number is", numbers[name])
```

check to see if the dictionary has a certain key and if it does prints out the number associated with it. Lastly if the menu choice is invalid it reprints the menu for your viewing pleasure.

A recap: Dictionaries have keys and values. Keys can be strings or numbers. Keys point to values. Values can be any type of variable (including lists or even dictionaries (those dictionaries or lists of course can contain dictionaries or lists themselves (scary right? :-)))).

Here is an example of using a list in a dictionary:

```
max_points = [25, 25, 50, 25, 100]
assignments = ['hw ch 1', 'hw ch 2', 'quiz ', 'hw ch 3', 'test']
students = {'#Max': max_points}

def print_menu():
    print("1. Add student")
    print("2. Remove student")
    print("3. Print grades")
    print("4. Record grade")
    print("5. Print Menu")
    print("6. Exit")

def print_all_grades():
    print('\t', end=' ')
    for i in range(len(assignments)):
        print(assignments[i], '\t', end=' ')
    print()
    keys = list(students.keys())
    keys.sort()
    for x in keys:
        print(x, '\t', end=' ') grades = students[x] print_grades(grades)

def print_grades(grades):
    for i in range(len(grades)):
        print(grades[i], '\t', end=' ')
    print()
```

END OF DEFINITIONS … PROGRAM IS ON NEXT PAGE

```
print_menu()
menu_choice = 0
while menu_choice != 6:
        print()
        menu_choice = int(input("Menu Choice (1-6): "))
        if menu_choice == 1:
                name = input("Student to add: ")
                students[name] = [0] * len(max_points)
        elif menu_choice == 2:
                name = input("Student to remove: ")
                if name in students:
                        del students[name]
                else:
                        print("Student:", name, "not found")
        elif menu_choice == 3:
                print_all_grades()
        elif menu_choice == 4:
                print("Record Grade")
                name = input("Student: ")
                if name in students:
                        grades = students[name]
                        print("Type in the number of the grade to record")
                        print("Type a 0 (zero) to exit")
                        for i in range(len(assignments)):
                                print(i + 1, assignments[i], '\t', end=' ')
                        print() print_grades(grades) which = 1234
                        while which != -1:
                                which = int(input("Change which Grade: "))
                                which -= 1        #same as which = which - 1
                                if 0 <= which < len(grades):
                                        grade = int(input("Grade: "))
                                        grades[which] = grade
                                elif which != -1:
                                        print("Invalid Grade Number")
        else:
                print("Student not found")
elif menu_choice != 6:
        print_menu()
```

Heres how the program works. Basically the variable `students` is a dictionary with the keys being the name of the students and the values being their grades. The first two lines just create two lists. The next line `students = {'#Max': max_points}` creates a new dictionary with the key `{#Max}` and the value is set to be `[25, 25, 50, 25, 100]` (since thats what `max_points` was when the assignment is made) (We use the key `#Max` since `#` is sorted ahead of any alphabetic characters). Next `print_menu` is defined. Next the `print_all_grades` function is defined.
Notice how first the keys are gotten out of the `students` dictionary with the `keys` function in the line `keys= list(students.keys())`. `keys` is a iterable, and it is converted to list so all the functions for lists can be used on it. Next the keys are sorted in the line `keys.sort()`. `for` is used to go through all the keys. The grades are stored as a list inside the dictionary so the assignment `grades = students[x]` gives `grades` the list that is stored at the key `x`. The function `print_grades` just prints a list and is defined a few lines later.

The later lines of the program implement the various options of the menu. The line `students[name] = [0] * len(max_points)` adds a student to the key of their name. The notation `[0] * len(max_points)` just creates a list of 0's that is the same length as the `max_points` list.

The `remove` student entry just deletes a student similar to the telephone book example. The record grades choice is a little more complex. The grades are retrieved in the line `grades = students[name]` gets a reference

to the grades of the student name. A grade is then recorded in the line grades[which] = grade. You may notice that gradesis never put back into the students dictionary (as in no students[name] = grades). The reason for the missing statement is that gradesis actually another name for students[name] and so changing gradeschanges student[name].

Dictionaries provide a easy way to link keys to values. This can be used to easily keep track of data that is attached to various keys.

Using Modules

Type this in (name it cal.py (importactually looks for a file named calendar.py and reads it in. If the file is named calendar.py and it sees a "import calendar" it tries to read in itself which works poorly at best.)):

```
import calendar

year = int(input("Type in the year number: "))
```

So what does the program do? The first line import calendaruses a new command import. The command importloads a module (in this case the calendarmodule). To see the commands available in the standard modules go to http://docs.python.org/3/library/. If you look at the documentation for the calendar module, it lists a function called prcalthat prints a calendar for a year. The line calendar.prcal(year) uses this function. In summary to use a module importit and then use module_name.function for functions in the module. Another way to write the program is:

```
from calendar import prcal

year = int(input("Type in the year number: "))
prcal(year)
```

This version imports a specific function from a module. Here is another program that uses the Python Library (name it something like clock.py) (press Ctrl and the 'c' key at the same time to terminate the program):

```
from time import time, ctime prev_time = ""
while True:
    the_time = ctime(time())
    if prev_time != the_time:
        print("The time is:", ctime(time()))
        prev_time = the_time
```

The program just does a infinite loop (Trueis always true, so while True:goes forever) and each time checks to see if the time has changed and prints it if it has. Notice how multiple names after the import statement are used in the line from time import time, ctime.

The Python Library contains many useful functions. These functions give your programs more abilities and many of them can simplify programming in Python.

Exercises

1. Rewrite the high_low.pyprogram from a previous learning guide to use a random integer between 0 and 1000 instead of a hard coded or operator input value. Use the Python documentation to find an appropriate module and function to do this.
2. Write a “craps” game which is a game using random rolls of 2 dice. The user bets on a roll of the dice. If the dice add to 7 or 11, the user wins. If the dice add to 2, 3, or 12, the user loses. Any other roll is called a “point” and the user bets of the following roll. If the “point” is rolled before a 7 is rolled, the user wins. If a 7 is rolled before the “point” is rolled again, the user loses.

More on Lists

We have already seen lists and how they can be used. Now that you have some more background we'll go into more detail about lists. First we will look at more ways to get at the elements in a list and then we will talk about copying them.

Here are some examples of using indexing to access a single element of a list:

```
>>> some_numbers = ['zero', 'one', 'two', 'three', 'four', 'five']
>>> some_numbers[0]
'zero'
>>> some_numbers[4]
'four'
>>> some_numbers[5]
'five'
```

All those examples should look familiar to you. If you want the first item in the list just look at index 0. The second item is index 1 and so on through the list. However what if you want the last item in the list? One way could be to use the `len()`function like `some_numbers[len(some_numbers) - 1]`. This way works since the `len()`function always returns the last index plus one. The second from the last would then be `some_numbers[len(some_numbers) - 2]`. There is an easier way to do this. In Python the last item is always `index -1`. The second to the last is `index -2` and so on. Here are some more examples:

```
>>> some_numbers[len(some_numbers) - 1]
'five'
>>> some_numbers[len(some_numbers) - 2]
'four'
>>> some_numbers[-1]
'five'
>>> some_numbers[-2]
'four'
>>> some_numbers[-6]
'zero'
```

Thus any item in the list can be indexed in two ways: from the front and from the back.

Another useful way to get into parts of lists is using **slicing**. Here is another example to give you an idea what they can be used for:

```
>>> things = [0, 'Fred', 2, 'S.P.A.M.', 'Stocking', 42, "Jack", "Jill"]
>>> things[0]
0
>>> things[7]
'Jill'
>>> things[0:8]
[0, 'Fred', 2, 'S.P.A.M.', 'Stocking', 42, 'Jack', 'Jill']
>>> things[2:4]
[2, 'S.P.A.M.']
>>> things[4:7]
['Stocking', 42, 'Jack']
>>> things[1:5]
['Fred', 2, 'S.P.A.M.', 'Stocking']
```

Slicing is used to return part of a list. The slicing operator is in the form `things[first_index:last_index]`. Slicing cuts the list before the `first_index`and before the `last_index`and returns the parts in between. You can use both types of indexing:

```
>>> things[-4:-2]
['Stocking', 42]
>>> things[-4]
'Stocking'
>>> things[-4:6]
['Stocking', 42]
```

Another trick with slicing is the unspecified index. If the first index is not specified the beginning of the list is assumed. If the last index is not specified the whole rest of the list is assumed. Here are some examples:

```
>>> things[:2]
[0, 'Fred']
>>> things[-2:]
['Jack', 'Jill']
>>> things[:3]
[0, 'Fred', 2]
>>> things[:-5]
[0, 'Fred', 2]
```

Here is a (HTML inspired) program example (copy and paste in the poem definition if you want):

```
poem = ["<B>", "Jack", "and", "Jill", "</B>", "went", "up", "the", "hill", "to", "<B>", "fetch", "a",
        "pail", "of", "</B>", "water.", "Jack", "fell", "<B>", "down", "and", "broke", "</B>", "his",
        "crown", "and", "<B>", "Jill", "came", "</B>", "tumbling", "after"]

def get_bolds(text):
        true = 1 false = 0
        ## is_bold tells whether or not we are currently looking at
        ## a bold section of text.
        is_bold = false
        ## start_block is the index of the start of either an unbolded
        ## segment of text or a bolded segment.
        start_block = 0
        for index in range(len(text)):
                ## Handle a starting of bold text
                if text[index] == "<B>":
                        if is_bold:
                                print("Error: Extra Bold")
                        ## print "Not Bold:", text[start_block:index]
                        is_bold = true
                        start_block = index + 1
                ## Handle end of bold text
                ## Remember that the last number in a slice is the index
                ## after the last index used.
                if text[index] == "</B>":
                        if not is_bold:
                                print("Error: Extra Close Bold")
                        print("Bold [", start_block, ":", index, "]", text[start_block:index])
                        is_bold = false
                        start_block = index + 1

get_bolds(poem)
```

The `get_bold()` function takes in a list that is broken into words and tokens. The tokens that it looks for are `<B>` which starts the bold text and `</B>` which ends bold text. The function `get_bold()` goes through and searches for the start and end tokens.

The next feature of lists is copying them. If you try something simple like:

```
>>> a = [1, 2, 3]
>>> b = a
>>> print(b)
[1, 2, 3]
>>> b[1] = 10
>>> print(b)
[1, 10, 3]
>>> print(a)
[1, 10, 3]
```

This probably looks surprising since a modification to `b` resulted in `a` being changed as well. What happened is that the statement `b = a` makes `b` a *reference* to `a`. This means that `b` can be thought of as another name for `a`. Hence any modification to `b` changes `a` as well. However some assignments don't create two names for one list:

```
>>> a = [1, 2, 3]
>>> b = a * 2
>>> print(a)
[1, 2, 3]
>>> print(b)
[1, 2, 3, 1, 2, 3]
>>> a[1] = 10
>>> print(a)
[1, 10, 3]
>>> print(b)
[1, 2, 3, 1, 2, 3]
```

In this case `b` is not a reference to `a` since the expression `a * 2` creates a new list. Then the statement `b = a * 2` gives `b` a reference to `a * 2` rather than a reference to `a`. All assignment operations create a reference. When you pass a list as an argument to a function you create a reference as well. Most of the time you don't have to worry about creating references rather than copies. However, when you need to make modifications to one list without changing another name of the list you have to make sure that you have actually created a copy.

There are several ways to make a copy of a list. The simplest that works most of the time is the slice operator since it always makes a new list even if it is a slice of a whole list:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b[1] = 10
>>> print(a)
[1, 2, 3]
>>> print(b)
[1, 10, 3]
```

Taking the slice `[:]` creates a new copy of the list. However it only copies the outer list. Any sublist inside is still a references to the sublist in the original list. Therefore, when the list contains lists, the inner lists have to be copied as well. You could do that manually but Python already contains a module to do it. You use the deepcopy function of the copy module:

```
>>> import copy
>>> a = [[1, 2, 3], [4, 5, 6]]
>>> b = a[:]
>>> c = copy.deepcopy(a)
>>> b[0][1] = 10
>>> c[1][1] = 12
>>> print(a)
[[1, 10, 3], [4, 5, 6]]
>>> print(b)
[[1, 10, 3], [4, 5, 6]]
>>> print(c)
[[1, 2, 3], [4, 12, 6]]
```

First of all notice that `a` is a list of lists. Then notice that `when b[0][1] = 10` is run both `a` and `b` are changed, but `c` is not. This happens because the inner arrays are still references when the slice operator is used. However with deepcopy `c` was fully copied.

So, should you worry about references every time you use a function or =? The good news is that you only have to worry about references when using dictionaries and lists. Numbers and strings create references when assigned but every operation on numbers and strings that modifies them creates a new copy so you can never modify them unexpectedly. You do have to think about references when you are modifying a list or a dictionary.

By now you are probably wondering why are references used at all? The basic reason is speed. It is much faster to make a reference to a thousand element list than to copy all the elements. The other reason is that it allows you to have a function to modify the inputed list or dictionary. Just remember about references if you ever have some weird problem with data being changed when it shouldn't be.