Variables homework

Advice for homework and life: try, play, experiment; don't Google until you get frustrated.

You can't break anything with Python, so, if you want to figure something out, try until you get it to work. You'll learn more and you'll remember it longer and more deeply.

In general, the homework question will consist of the question in a markdown cell, a code cell for you to play in, repeating as needed, and a final markdown cell for your answer or explanation *that will be in italics*.

But do feel free to add or delete cells as you feel appropriate. The important thing is you get your point across!

1.

Make an integer: `theAnswer = 42` . Now make another: `anotherNameForTheAnswer = 42` (You don't have to use these exactly, but make sure you have two different names referring to the same exact value.)

In [1]:
```
1 theAnswer = 42
2 anotherNameForTheAnswer = 42
```

Get and note the ID numbers for both.

In [2]:
```
1 print(f"ID of 'theAnswer' = {id(theAnswer)}")
2 print(f"ID of the 'anotherNameForTheAnswer' = {id(anotherNameForTheAnswer)}")
```

```
ID of 'theAnswer' = 2794969067088
ID of the 'anotherNameForTheAnswer' = 2794969067088
```

Assign each name to a completely different number and confirm that each name now refers to an object with a new ID.

In [3]:
```
1 theAnswer = 43
2 anotherNameForTheAnswer = 39
3 print(f"ID of 'theAnswer' = {id(theAnswer)}")
4 print(f"ID of the 'anotherNameForTheAnswer' = {id(anotherNameForTheAnswer)}")
```

```
ID of 'theAnswer' = 2794969067120
ID of the 'anotherNameForTheAnswer' = 2794969066992
```

Do a `whos` to confirm that *no* names refer to the original value.

In [4]:
```
1 whos
```

```
Variable                 Type    Data/Info
-----------------------------------------
anotherNameForTheAnswer  int     39
theAnswer                int     43
```

Finally, assign a new name to the original value, and get its ID.

In [6]:
```
1 newName = 42
2 print(f"This is the ID of newName, which was assigned to the old object of 42: {id(newName)}")
```

```
This is the ID of newName, which was assigned to the old object of 42: 2794969067088
```

Look at this ID and compare it to the original ones. What happened? What does this tell you?

The id of newName is the same as the original id of theAnswer. This tells me that ids are assigned to objects, and they remain the same regardless of what names are associated with that object. The id is not associated with the names I create, but rather the objects themselves.

2.

Convert both possible Boolean values to strings and print them.

In [2]:
```
convertFalseToStr = str(False)
convertTrueToStr = str(True)

convert0ToStr = str(int(False))
convert1ToStr = str(int(True))

print(f"This is what 'False' looks like when it's converted to a string: {convertFalseToStr}")
print(f"This is what 'True' looks like when it's converted to a string: {convertTrueToStr}")

print(f"This is what '0' looks like when it's converted to a string: {convert0ToStr}")
print(f"This is what '1' looks like when it's converted to a string: {convert1ToStr}")
```

```
This is what 'False' looks like when it's converted to a string: False
This is what 'True' looks like when it's converted to a string: True
This is what '0' looks like when it's converted to a string: 0
This is what '1' looks like when it's converted to a string: 1
```

Now convert some strings to Boolean until you figure out "the rule".

In [10]:
```
str1 = "False"
bool(str1)

str2 = "True"
bool(str2)

str3 = "0"
bool(str3)

str4 = "4"
bool(str4)

str5 = " "
bool(str5)

str6 = ""
bool(str6)
```

Out[10]:
```
False
```

What string(s) convert to True and False? In other words, what is the rule for str -> bool conversion?

Any non-empty string converts to True and an empty string converts to False

---

3.

Make three variables:

- a Boolean equal to True
- an int (any int)
- a float

Try all combinations of adding two of the variables pairwise.

In [37]:
```
boolVar = True
intVar = 5
floatVar = 3.6
```

In [39]:
```
print(boolVar + intVar)
print(f"This is the new type when adding bool and int: {type(boolVar + intVar)}")
```

```
5
This is the new type when adding bool and int: <class 'int'>
```

In [32]:
```
print(boolVar + floatVar)
print(f"This is the new type when adding bool and float: {type(boolVar + floatVar)}")
```

```
4.6
This is the new type when adding bool and float: <class 'float'>
```

In [34]:
```
print(intVar + floatVar)
print(f"This is the new type when adding int and float: {type(intVar + floatVar)}")
```

```
8.6
This is the new type when adding int and float: <class 'float'>
```

In [35]:
```
print(boolVar + boolVar)
print(f"When adding a bool to bool, the new type is: {type(boolVar + boolVar)}")
```

```
2
When adding a bool to bool, the new type is: <class 'int'>
```

What is the rule for adding numbers of different types?

- When adding a boolean value to a number, the boolean value will be converted to a 1 or 0
- When adding 2 numbers, the new number will be a float if one of the original numbers is a float otherwise the new value will be an integer

---

4.

Make an int (any int) and a string containing a number (e.g. num_str = '64'). Try

- adding them
- adding them converting the number to a string
- adding them converting the string to a number

In [40]:
```
num_int = 2
num_str = "23"
```

In [41]:
```
num_int + num_str
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_17588/3818996895.py in <module>
----> 1 num_int + num_str

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In [42]:
```
str(num_int) + num_str
```

Out[42]: '223'

In [43]:
```
num_int + int(num_str)
```

Out[43]: 25

Try converting a `str` that is a spelled out number (like 'forty two') to an int.

In [45]:
```
new_str = 'twelve'
int(new_str)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_17588/3723376432.py in <module>
      1 new_str = 'twelve'
----> 2 int(new_str)

ValueError: invalid literal for int() with base 10: 'twelve'
```

Did that work?

It didn't work to convert a spelled out number that's formatted as a string to an integer

---

5.

Make a variable that is a 5 element tuple.

In [46]:
```
1 myTuple = (1, 2, 3, 4, 5)
```

Extract the last 3 elements.

In [48]:
```
1 myTuple[-3:]
```

Out[48]: (3, 4, 5)

6.

Make two variables containing tuples (you can create one and re-use the one from #5). Add them using "+".

In [50]:
```
1 myTuple = (1, 2, 3, 4, 5)
2 anotherTuple = (6, 7, 8)
3 myTuple + anotherTuple
```

Out[50]: (1, 2, 3, 4, 5, 6, 7, 8)

In [53]:
```
1 myTuple = (1, 2, 3, 4, 5)
2 anotherTuple = (6, 7, 8, 9, 10)
3 myTuple + anotherTuple
```

Out[53]: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Make two list variables and add them.

In [51]:
```
1 myList = [1, 2, 3, 4, 5]
2 anotherList = [6, 7, 8]
3 myList + anotherList
```

Out[51]: [1, 2, 3, 4, 5, 6, 7, 8]

In [52]:
```
1 myList = [1, 2, 3, 4, 5]
2 anotherList = [6, 7, 8, 9, 10]
3 myList + anotherList
```

Out[52]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Try adding one of your tuples to one of your lists.

In [56]:
```
1 myList + anotherTuple
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_17588/2818925830.py in <module>
----> 1 myList + anotherTuple

TypeError: can only concatenate list (not "tuple") to list
```

What happened? How does this compare to adding, say, a bool to a float?

Adding a tuple and a list is different from adding a bool to a float because python doesn't convert either of the data types to match the other, making it impossible to add a tuple and a list.

7.

In [58]:
```
1 myTuple
```

Out[58]: (1, 2, 3, 4, 5)

Can you tell the type of a variable by looking at its value?

Yes, I can tell the type of variable by looking at its value because each type has its own unique characteristic

If so, how? A couple examples are fine; no need for an exhaustive list.

- A bool is always True or False
- An integer is always a whole number without decimal places
- A float is always a number with a decimal point
- A string is always enclosed in quotations
- A tuple is always enclosed by parentheses
- A list is always enclosed with brackets
- A dictionary is enclosed by curly braces and uses colons

8.

Make a list variable in which one of the elements is itself a list (e.g. `myList = ['hi', [3, 5, 7, 11], False]`).

In [59]:
```
1 myList = ['bonjour', [2, 4, 6, 8], True]
```

Extract one element of the nested list - the list-within a list. Try it in two steps, by first extracting the nested list and assigning it to a new variable.

In [60]:
```
1 listWithinList = myList[1]
2 listWithinList[3]
```

Out[60]: 8

Now see if you can do this in one step.

In [64]:
```
1 myList[1][3]
```

Out[64]: 8

9.

Make a `dict` variable with two elements, one of which is a list.

In [65]:
```
1 myDict = {'element1':1, 'element2':[2,3,4]}
```

Extract a single element from the list-in-a-dict in one step.

In [76]:
```
1 myDict['element2'][0]
```

Out[76]: 2

10.

Make a list variable. Consider that each element of the list is logically an *object* in and of itself. Confirm that one or two of these list elements has its own unique ID number.

In [ ]:
```
myList = ['bonjour', [2, 4, 6, 8], True]
```

In [78]:
```
id(myList)
```

Out[78]: 2217498013312

In [77]:
```
id(myList[0])
```

Out[77]: 2217498834992

In [79]:
```
id(myList[1])
```

Out[79]: 2217498009792

In [80]:
```
id(myList[1][1])
```

Out[80]: 2217417271696

In [81]:
```
id(myList[2])
```

Out[81]: 140727470610536

If you extract an element from your list and assign it to a new variable, are the IDs the same, or is a new object created?

In [82]:
```
newVar = myList[1][1]
id(newVar)
```

Out[82]: 2217417271696

Are the IDs the same, or is a new object created when you assigned the list element to new variable?

The IDs are the same when I assign a list element to a new variable

---

11.

Make a `str` variable containing the first 5 letters of the alphabet (e.g. `a2e = 'abcde'`). Check the ID of the second (index = 1) element (the 'b').

In [84]:
```
a2e = 'abcde'
id(a2e[1])
```

Out[84]: 2217418109424

Now make a `str` variable containing the letter 'b'. Check its ID.

In [86]:
```
newStr = 'label'
id(newStr[2])
```

Out[86]: 2217418109424

What happened?

B has the same id across all the strings that it is in

Each character within a string has its own unique id