

Name: _____

Partner: _____

Python Activity 19: Identity & Value

How do variables actually work?

Learning Objectives

Students will be able to:

Content:

- Describe what the built-in function *id(.)* does
- Explain the difference between *identity* and *value*
- Summarize how *memory address* relates to *mutability*
- Describe what the *is* and *+=* operators do

Process:

- Write code that uses the *is* operator to *appropriately* compare objects
- Write code to *mutate* a mutable object
- Write code to determine what other *object* types are mutable

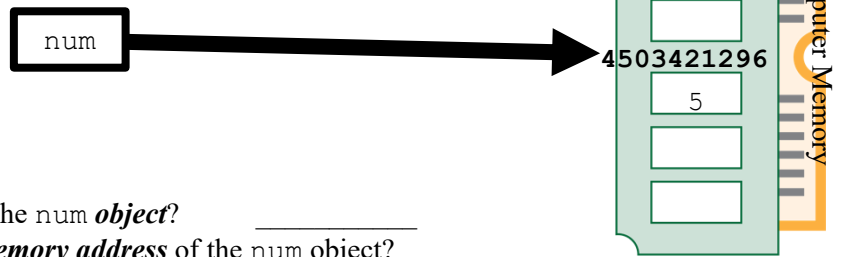
Prior Knowledge

- Python concepts: lists, strings, boolean operators

Critical Thinking Questions:

1. Examine the sample interactive python interaction and diagram:

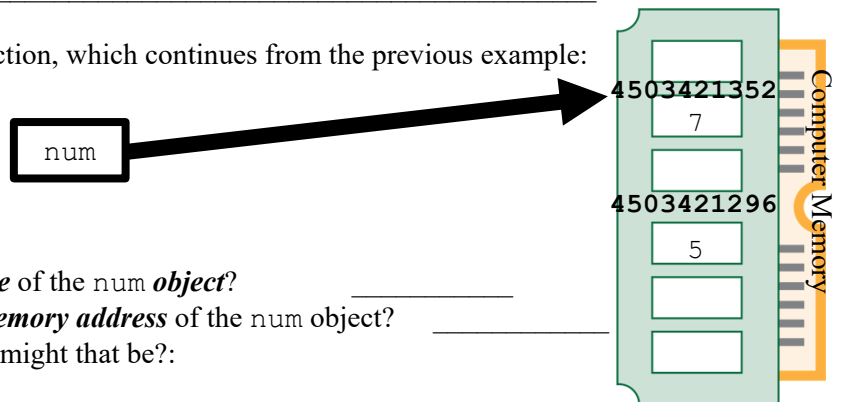
```
Interactive Python
>>> num = 5
>>> id(num)
4503421296
```



- What is the *value* of the *num object*? _____
- What might be the *memory address* of the *num object*? _____
- What might the built-in function *id(.)* do?:

2. Examine the following interaction, which continues from the previous example:

```
Continued
>>> num = num + 2
>>> id(num)
4503421352
```



- What is now the *value* of the *num object*? _____
- What might be the *memory address* of the *num object*? _____
- What changed? Why might that be?:

FYI: Everything in Python is an *object*. When an object is created, it is assigned an address in memory (or *identity*). An object's identity never changes. An object also has a *value*. Objects whose value *can* change are *mutable*. Objects whose value *cannot* change are *immutable*. *Variable* names *point* to memory addresses of a stored value.

3. Observe the following interaction in interactive python:

```
0 >>> plants1 = ["jade", "aloe", "fig"]
1 >>> id(plants1)
2 4336302400
3 >>> plants2 = plants1
4 >>> id(plants2)
5 4336302400
6 >>> plants1 == plants2
7 True
8 >>> plants1 is plants2
9 True
```

a. How do the identities of plants1 and plants2 compare?



What might the plants2 = plants1 line do?

How does it affect the *memory address* of plants2?

b. What values do plants1 and plants2 point to?



Why does the plants1 == plants2 line return its boolean value?

FYI: The values outputted by the *id(.)* function can change between python sessions, or on one machine versus another. Therefore, the *value* is not the important part of *id()*, just whether two *id()*s match!

4. Observe the following interaction in interactive python:

```
>>> plants1 = ["jade", "aloe", "fig"]
>>> id(plants1)
4336302400
>>> plants2 = ["jade", "aloe", "fig"]
>>> id(plants2)
4336284224
>>> plants1 == plants2
True
>>> plants1 is plants2
False
```




a. How do the identities of plants1 and plants2 compare in this example?

What did we do differently in this example, compared to the previous one?

How might this have impacted the result of `plants1 is plants2`?

What does this suggest about the `is` operator?

b. Why does the `plants1 == plants2` line return its boolean value?


 c. What does the above code tell us about the difference between the `==` operator and the `is` operator?

FYI: The *is* operator essentially uses the *id(..)* function to compare the *addresses* of two objects. The `==` operator compares their *values*.

5. Observe the following session in interactive python:

```
0 >>> plant = "spider"
1 >>> plant[-1] = "y"
2 TypeError: 'str' object does not support item assignment
3 >>> plants = ["jade", "aloe", "fig"]
4 >>> plants[-1] = "cactus"
```

a. What is the programmer trying to do on lines 0 & 1? (*Hint: what appears on the lefthand side of an assignment operator? The righthand side?*)

 b. Why might line 1 cause an error for string `plant`, but not line 4 with list `plants`?

c. What is likely contained in list `plants` at the end of this code?

FYI: *Mutable* objects are *changeable*. Lists can be *changed* using **indexing**, which means we can put an indexed list on the *lefthand* side of an assignment operator and reassign its values!

d. Are **strings** mutable? How do you know?

6. Observe the following session in interactive python:

```
>>> plants1 = ["jade", "aloe", "fig"]
>>> plants2 = plants1
```

a. Draw a diagram, similar to questions 1 & 2 that shows the two variables above, pointing to their values in memory:

- b. Modify your diagram above to incorporate this interaction:

```
>>> plants1 = plants1 + ["pothos"]
>>> plants1 is plants2
False
```

- c. What might `plants1` now point to? `plants2`?



- d. What change might you expect to see in the id of `plants1` and `plants2`, when reassigning `plants1`?

FYI: *Concatenation* + always returns a new object.

7. Observe the following session in interactive python:

```
>>> plants1 = ["jade", "aloe", "fig"]
>>> plants2 = plants1
>>> plants1 += ["zz"]
>>> plants1
['jade', 'aloe', 'fig', 'zz']
>>> plants2
['jade', 'aloe', 'fig', 'zz']
```

- a. Circle the new **operator** being used in this example, that wasn't in the previous example.
b. Draw a diagram, similar to questions 1 & 2 that shows the two variables above, pointing to their values in memory at the end of the code:



- c. How does the `+` operator differ from the `+=` operator, with mutable objects (like lists)?

FYI: *Mutable* objects are *changeable*. Lists can be *changed* using the **append operator** `+=`, **indexing**, and other methods we will learn about later.

Application Questions: Use the Python Interpreter to check your work

1. Write and execute code to determine if the following *object types* are mutable or not:

Lists: _____

Strings _____

Integers _____

Floats: _____

Booleans: _____

2. *This example/information will not be on any exam or homework!*


Observe the following interaction in interactive python:

```
>>> tea1 = "assam"
>>> id(tea1)
4462577328
>>> tea2 = "assam"
>>> id(tea2)
4462577328
>>> tea1 == tea2
True
>>> tea1 is tea2
True
```

a. Why does the `tea1 == tea2` line return its boolean value?

b. Why might the `tea1 is tea2` line return its boolean value?

c. What does the above code and previous example tell us about the difference between strings and lists?

 d. What does the above code and previous example tell us about the difference between immutable and mutable objects when creating new values?

FYI: Python has a *confusing* optimization where small numbers (-5...255) are remembered in an *integer pool* and short strings (less than 20 characters and created at compile-time and lack special characters) are *interned*, so these values created separately will point to the same memory address. With strings, Python may intern more than just the specified strings above.
Do not rely on this behavior!