National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester II, 2024/2025

Additional Reading
Aliasing

This document provides an additional discussion on the topic of aliasing. Some material here extends beyond the scope of CS1010S, but it offers a more complete understanding of the topic.

If you find this content confusing or overwhelming, feel free to skip it.

What is Aliasing?

Aliasing is when multiple names (variables) refer to the same object in memory. This occurs because Python variables store references to objects rather than the objects themselves. When an alias is created, modifying the object through one name affects all aliases that point to it.

This is a simple example of aliasing:

```
a = [1, 2, 3]
b = a

b[0] = 4
print(a) # [4, 2, 3]
print(b) # [4, 2, 3]
```

You can see that changing the first element of `b` also changes the first element of `a`. This is because `a` and `b` are aliases of the same list object.

This is also aliasing, but less trivial:

```
a = [1, 2, 3]

def f(x):
    x[0] = 4

f(a)
print(a) # [4, 2, 3]
```

In this case, passing `a` as an argument to the function `f` creates an alias `x`, which allows modifications to persist outside the function.

What causes Aliasing?

If you are looking for simple answer to this question, it is because of references and mutability, and draw box and pointer diagrams will help you understand it. Function's arguments are passed by reference, meaning that the function receives a reference to the object, not a copy of the object. This is why changes to the object persist outside the function. (THE END :D)

If you are looking for a more detailed answer, read on.

Aliasing is more complicated than it seems. Here, I offers a naive explanation of aliasing. In reality, aliasing is a result of the way Python manages memory and the trade-offs it makes to support mutable variables.

How variables is stored in python? How variables/input is passed to functions?

In Python, everything is an object. Variables store references to objects, not the objects themselves. When you assign a variable to an object, you are storing a reference to that object in memory.

Here is an example:

```
a = 1
b = a
id(a) # 140732674000000
id(b) # 140732674000000

c = [1,2,3]
id(c) # 140736000000000
```

Note that the id values could be difference in your case.

When you pass a variable to a function, you are passing a reference to the object that variable refers to. This is called pass-by-object-reference. This means that the function receives a reference to the object, not a copy of the object.

Let's verify this with an example:

```
>>> a = [1, 2, 3]
>>> def f(x):
      print(id(x))
      a[0] = 4
      print(id(x))

>>> id(a)
140736000000000 # id of a
>>> f(a)
140736000000000 # same id implies same object
140736000000000 # same id implies same object
>>> print(a)
[4, 2, 3] # a is modified
```

So, we can see that the id of `a` and `x` are the same, which means that they refer to the same object in memory. This is why changes to `x` persist outside the function.

Why doesn't tuple have aliasing issues?

Although, tuples is also a reference type and pass-by-object-reference into functions. However, tuples are immutable, which means that they cannot be changed after they are created, so aliasing is not an issue with tuples.

This following example demonstrates that aliasing is not an issue with tuples because Python will stop you from modifying the tuple (Python is screaming):

```
>>> a = (1, 2, 3)
>>> b = a
>>> id(a)
140736000000000
>>> id(b)
140736000000000
>>> def f(x):
      print(id(x))
      x[0] = 2 # THIS IS A CRIME, YOU SHOULD SCREAM WHEN YOU SEE THIS
      print(id(x))

>>> f(a)
140736000000000 # id of a
  ...
TypeError: 'tuple' object does not support item assignment
```

Figure 1: This is a crime, you should scream.

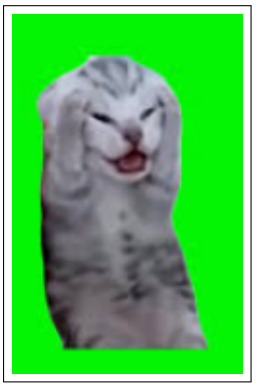

Figure 2: Zhu Ming when he saw his student(s) modifying tuples in the PE.

But you can reassign the tuple to a new tuple, which is not a crime:

```
>>> a = (1, 2, 3)
>>> b = a
>>> id(a)
140736000000000
>>> id(b)
140736000000000

>>> def f(x):
      print(id(x))
      x = (4, 2, 3) # reassing x to a new tuple
      print(id(x))

>>> f(a)
140736000000000 # id of a
140736000000016 # new id implies new object
```

How about other data types like interger?

From your experience, you know that aliasing is not an issue with integers. But why? Can you explain why aliasing is not an issue with integers?

You can try to above examples with integers and see what happens.

Conclusion

Aliasing is a result of:

- References: Variables store references to objects, not the objects themselves.
- Mutability: Objects can be changed after they are created.
- Pass-by-reference: Functions receive references to objects, not copies of objects.

When multiple variables share a reference to a mutable object (through function calls, assignments, etc.), aliasing issues arise.

All can be easily understood by drawing box and pointer diagrams.