Memory and References

- PoPI

Memory Partition: Stack and Heap

Amazon Locker

Amazon Warehouse

Stack

- Access to items is quick
- Only small items

Heap

- Access to items is slow
- Large items

Delivering Items

- If a small item needs to be delivered to A

  - place it in the locker A

- If a large item need to be delivered to A

???

  - Place the large item to the warehouse
    - To the first available spot, e.g., warehouse X, compartment N, shelf T
  - Place a note (reference) in the locker A saying
    - "Your item can be found in warehouse X, compartment N, shelf T"

Datatypes and their storage

- Each variable A, B,… mentioned in a Python program gets a “box” allocated on the stack
- The values of those variables are stored differently depending on variable’s datatype
- Values for lighter datatypes are stored on the stack
  - Integers, strings, floats, characters, Booleans, ….
- Values for heavier datatypes are stored on the heap
  - with references to the values stored on the stack
  - Lists, dictionaries, sets, classes, ….
- Note: this separation varies for programming languages and even for different implementations of the same language
- Demo: storing integers, lists, strings, etc. in Python https://goo.gl/ATBDHQ

Aliasing

- Consider the example:

```
b = 10
c = b
b = 5
print( c)
```

Result?

```
>> 10
```

```
B = ["MacBook", "Toaster", "Toilet Paper"]
C = B
B[0] = "PC"
print(C)
```

Result?

```
>> ["PC", "Toaster", "Toilet Paper"]
```

Demo: https://goo.gl/VssgEd

Aliasing (cont.)

- We may need C to refer to a copy of the object B refers to (instead of the object itself)

```
B = [“MacBook”, “Toaster”, “Toilet Paper”]
C = B[:]
B[0] = “PC”
print(C)
```

Result?

```
>> [“MacBook”, “Toaster”, “Toilet Paper”]
```

Demo: https://goo.gl/R8fp5j

Alternatively:

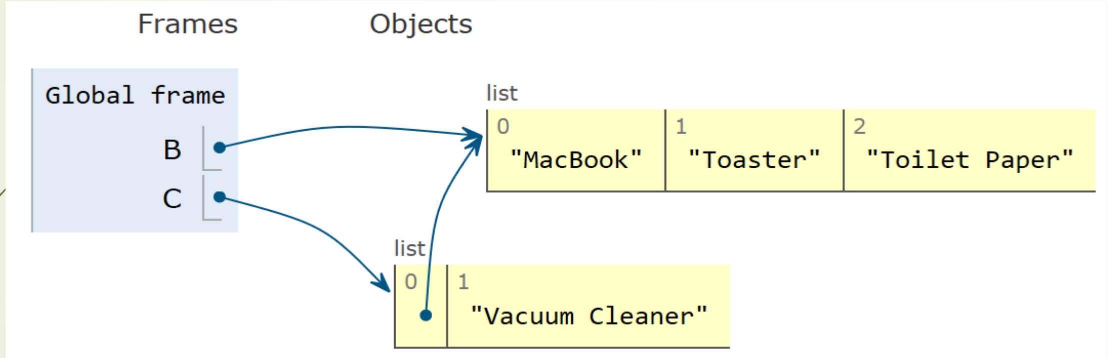
```
import copy
…
C = copy.copy(B)
```

"Double" Referencing

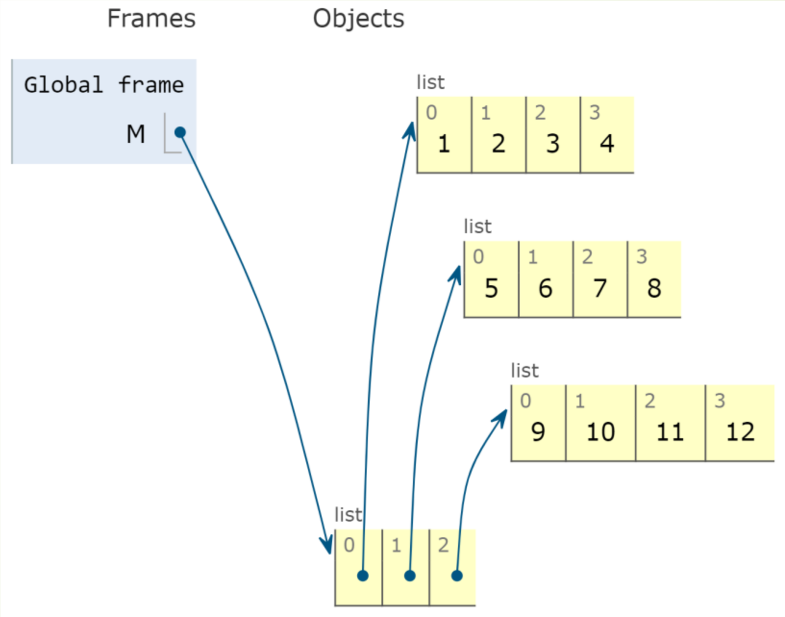
- Values on the heap can store references to other values on the heap

B = ["MacBook", "Toaster", "Toilet Paper"]

C = [B, "Vacuum Cleaner"]



M = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]



- Note: values that are not referenced from anywhere are lost - demo https://goo.gl/y7hsCb

Functions and reference types

Passing Arguments to Functions

- Consider two very similar programs:
  - 1)
    ```
    def reduce_by_1(n):
            n = n-1
    A = 5
    reduce_by_1(A)
    print(A)
    ```

    Result?

    a) 4

    b) 5

    c) “Toilet paper”

  - 2)
    ```
    def reduce_by_1(pair):
            pair[0] = pair[0]-1
            pair[1] = pair[1]-1
    A = [5,20]
    reduce_by_1(A)
    print(A)
    ```

    Result?

    a) [4,19]

    b) [5,20]

    c) [5,21]

- Demo: second program execution

Passing Arguments to Functions (cont.)

- For any variable A, when a function fun(n) is called on it:
  - a new variable n is created on the stack
  - the stack content of A is copied to the stack content of n
  - Therefore:
    - If A has a lighter datatype, the value of A itself is copied to n
    - If A has a heavier datatype, the reference to the value of A is copied to n
  - The first mechanism of passing arguments is call by value
  - The second mechanism is call by reference
- Bottom line:
  - Functions are called by value on arguments that are: integers, floats, strings,…
  - Functions are called by reference on arguments that are: lists, dictionaries, sets,…

Returning function results

- What we have said about passing arguments to function applies to returning results from functions
- If R is the result to be returned from function fun(n) and A = fun(n)
  - if R has a lighter datatype, then the value of R itself is copied to A
  - If R has a heavier datatype, then the reference to the value of R is copied to A