Interview Cake

← course home (/table-of-contents#section_general-programming_concept_garbage-collection)

Garbage Collection

A **garbage collector** automatically frees up memory that a program isn't using anymore.

For example, say we did this in Python 2.7:

Python 2.7

```
def get_min(nums):
    # Note: this is *not* the fastest way to get the min!
    nums_sorted = sorted(nums)
    return nums_sorted[0]

my_nums = [5, 3, 1, 4, 6]
print get_min(my_nums)
```

Look at `nums_sorted` in `get_min()`. We allocate that whole list inside our function, and once the function returns we don't need the list anymore. In fact, once the function returns we *don't have any references to it anymore*!

What happens to that list in memory? The Python 2.7 garbage collector will notice we don't need it anymore and free up that space.

> How does a garbage collector know when something can be freed?
>
> One option is to start by figuring out what we *can't* free. For example, we definitely can't free local variables that we're going to need later on. And, if we have a list, then we also shouldn't free any of the list's elements.
>
> This is the main intuition behind one garbage collector strategy:
>
> 1. Carefully figure out what things in memory we might still be using or need later on.
> 2. Free everything else.

> This strategy is often called **tracing garbage collection**, since we usually implement the first step by tracing references from one object (say, the list) to the next (an element within the list).
>
> A different option is to have each object keep track of the number of things that reference it—like a variable holding the location of an array or multiple edges pointing to the same node in a graph. We call this number an object's **reference count**.
>
> In this case, a garbage collector can free anything with a reference count of zero.
>
> This strategy is called **reference counting**, since we are *counting* the number of times each object is *referenced*.

Some languages, like C, don't have a garbage collector. So we need to manually free up any memory we've allocated once we're done with it:

C

```
// make a string that can hold 15 characters
// including the terminating null byte ('\0')
str = malloc(15);

// ... do some stuff with it ...

// we're done. free that memory!
free(str);
```

We sometimes call this **manual memory management**.

Some languages, like C++, have both manual and automatic memory management.

← course home (/table-of-contents)

Next up: Closures → (/concept/js-closure?course=fc1&section=general-programming)