Memory

Intro

Our Goals

- Overview of computer memory
- Variables and identifiers
- Mutable versus immutable objects
- How this works with Python lists

Why Do We Care?

```
>>> a = ['hello', 'there', 'hackbright']
>>> b = a
>>> a.append('yay')
```

- Does ***b*** have `'yay'` in it?
- How could you know without checking?

Physical Memory

Memory

Can think of computer memory like a sheet of graph paper:

	0	1	2	3	4	5	6	7	8	9
0										
1										

- Each spot is the same size
- We can refer to each spot by number
    - For example: y=1, x=2 as `$12`

Bits and Bytes

Bit: 2 possibilities (0 | 1)

0/1

2 Bits: 4 possibilities (00 | 01 | 10 | 11)

0/1	0/1

8 Bits (“byte”): 256 possibilities (2^8)

0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1

What Fits?

- Tiny numbers *(0-255)*: 1 byte
- Integers *(≈ 4.3 billion)*: 4 bytes
- Floating point numbers: 8 bytes
- Strings: 1 byte per character

Note: The statements above aren't always true

On older computers, integers were often represented with 2 bytes, so they could range from 0—65,535. A differently-named structure, a "long integer," was used to hold numbers up to ≈ 4.3 billion (made up of, yes, 4 bytes). These kind of differences are mostly historic and not particularly important to programmers in higher-level languages like Python.

More pertinent to current times, but still a bit obscure for this lecture, is that strings are made of "unicode symbols" that are bigger than 1 byte. This would most often be when accents or unusual symbols are part of the string: So "Héllö" could be made of up 5 characters but might take 9 bytes to store.

For now, we'll only worry about strings where 1 character is equal to 1 byte.

To learn more read about strings.

Memory

	0	1	2	3	4	5	6	7	8	9
0		int:	42		h	e	l	l	o	
1										

- $00-03 contains `42`
- $04 contains `h`

A Note on Python Strings

Python strings use an encoding system called unicode allowing a wider range of characters.

For some operations, Python will utilize a bytestring — a string represented in bytes.

Bytestrings are denoted with a ‘b’.

```
>>> string = 'I am a string'
>>> type(s)
<class 'str'>
>>> byte_string = b'I am a string'
<class 'bytes'>
```

Note: You’ll encounter bytestrings later

This information will come in handy when we show you how to write tests for your web application.

Variables

Way Old Skool: Assembly Language

```
LDA $0607;
INC;
STA $0607;
```

- Load value stored at `$0607`
- Increment it
- Write (store) the value to `$0607`

Classic Variables: C

Classic "variable": a fixed place in memory

```
void add_together(int x, int y) {
    int z;

    z = x + y;
}
```

- ***x***, ***y***, and ***z*** are fixed places in memory
- We know how large they are
- They never move around

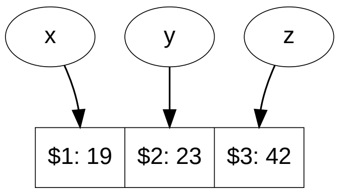
Note: Compiled languages

With this style of languages, you have to declare what kind of information a variable held (this is an integer, this is a 25-character-long string, this is a float, etc) because the computer would set aside exactly enough memory, and would "point" the name to that spot in memory.

For example, since ***x*** in this function is an integer, on most computers, 4 bytes of memory would be set aside and the name ***x*** would point to that spot in memory. If you updated ***x*** with a line like `x = 17`, it would keep ***x*** as a name pointing to that spot in memory, and update the information directly there. The location pointed to by the name ***x*** never changes.

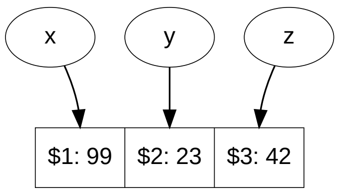
Locations in Memory in C

```
x = 19;     // x permanently points to mem location $1
y = 23;     // y to $2
z = 42;     // z to $3
```



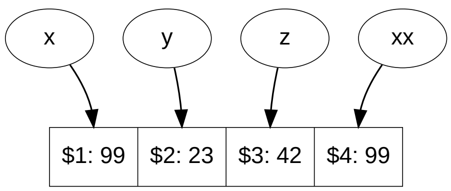
Then, a line of code runs that changes ***x***:

```
x = 99;   // change data stored in x (location $1)
```



Then we create ***xx*** and assign it the same values as ***x***:

```
xx = x;   // xx permanently points to $4 (and gets value x had)
```

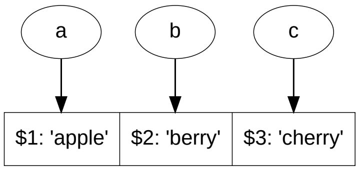


Python Doesn’t Have Variables

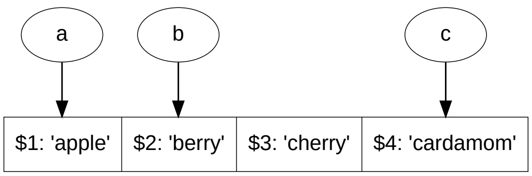
- Not in the classic sense
- It has “identifiers”
  - A name that points to a spot in memory
  - But the name can **move**
- Yes, you can continue to call them variables :)

Python Memory

```
a = 'apple'
b = 'berry'
c = 'cherry'
```




```
c = 'cardamom'
```



- `=` in Python doesn't mean "change memory to this value"
- It means: "bind this name to this value"

Immutable / Mutable

Immutable Types

- Immutable: can't change
  - Strings, Numbers, Tuples (& more)

```
msg = 'hello'

msg = 'hi'      # ok! rebinds

msg[0] = 'H'    # error: immutable!

msg = 'Hi'      # ok! rebinds
```

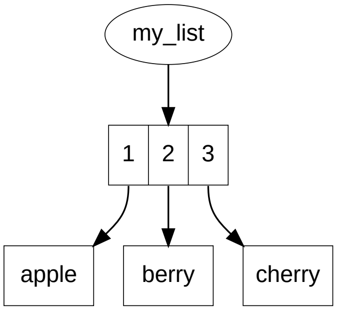
Mutable Types

- Mutable: can change
  - Lists, dictionaries, sets, objects (& more)

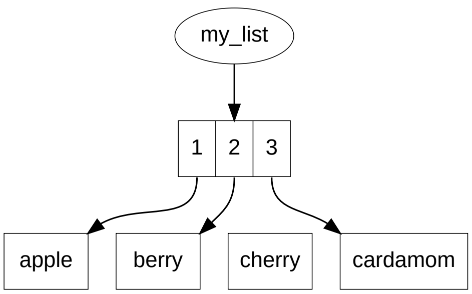
```
my_list = ['apple', 'berry', 'cherry']

my_list[2] = 'cardamom'     # ok! mutable

my_list = ['apple', 'berry', 'cardamom']    # ok, rebinds!
```

Lists

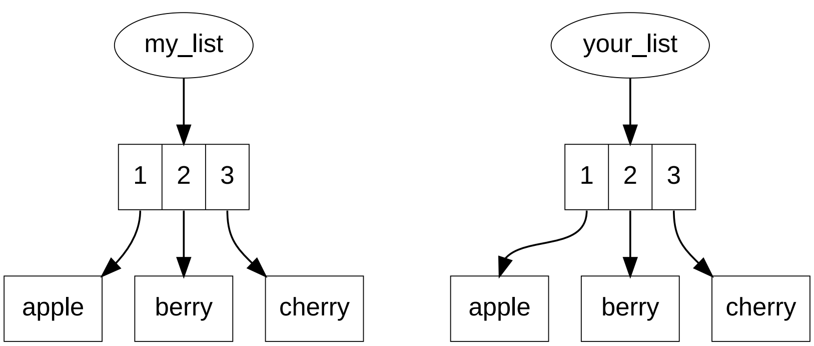
```
my_list = ['apple', 'berry', 'cherry']
```



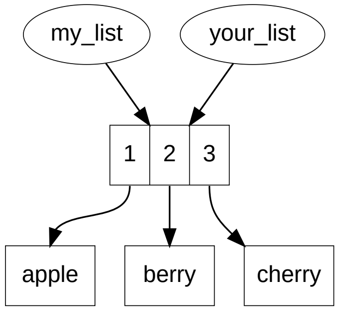

```
my_list = ['apple', 'berry', 'cherry']
my_list[2] = 'cardamom'
```




```
my_list = ['apple', 'berry', 'cherry']
your_list = ['apple', 'berry', 'cherry']
```




```
my_list = ['apple', 'berry', 'cherry']
your_list = my_list
```



But That’s Not What I Wanted!

- You can copy the list—not just bind a new name

```
my_list = ['apple', 'berry', 'cherry']

from copy import copy
your_list = copy(my_list)
```

- Or, you can (ab)use list slices

```
your_list = my_list[:]
```

Identity

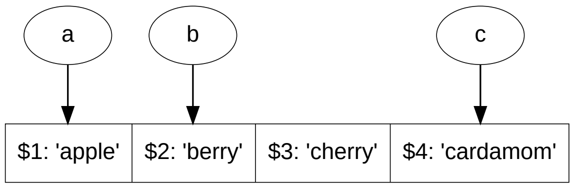
id() and is

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

>>> id(a)
99999

>>> id(b)
12345

>>> a == b
True

>>> a is b
False
```

- `==` checks “equality,” `is` checks “identity”

Cleaning Up

```
a = 'apple'
b = 'berry'
c = 'cherry'

c = 'cardamom'
```



- What happens to “cherry” ?

Garbage Collection

- After a while, Python notices nothing points to “cherry”
  - And it will “release” that memory
- This happens when program finishes
  - And often during the program, but you don’t know when

What Did We Learn?

- Variables in Python can be of any type
  - And can change during the program run
- Read `=` as “binding” (“drawing a new arrow”)
- `x = y` means “bind ***x*** to whatever ***y*** is bound to”

Looking Ahead

- Compiled vs. interpreted languages
- Way later in the course: how lists, sets, and dictionaries are implemented

© 2020 Hackbright Academy