Genome 540 discussion

January 9th, 2025
Joe Min

Genome Sciences
UNIVERSITY OF WASHINGTON

Agenda

Memory and pointers

Getting started in C++

Getting around Python

Memory and pointers

Computers handle data like a post office

Permanent things, like files, get saved “to disk”, or are given a permanent home in the mail room

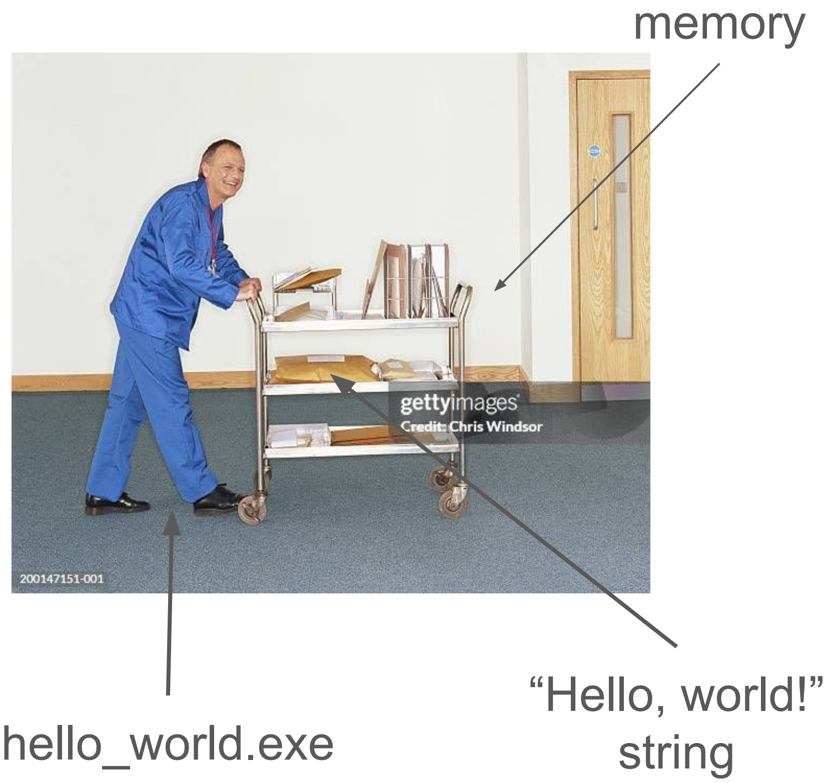
540_slides_final.pptx

Korea trip photos

Memory is a storage cart

Running programs, on the other hand, are more like office staff with a shared cart, or RAM

Staff can temporarily store, move, and change things on the shared cart, but only the parts allocated to them



memory

hello_world.exe

“Hello, world!” string

Packages hold data

Packages are the data that take up space on the cart, such as variables in a program (e.g., ints and strings)

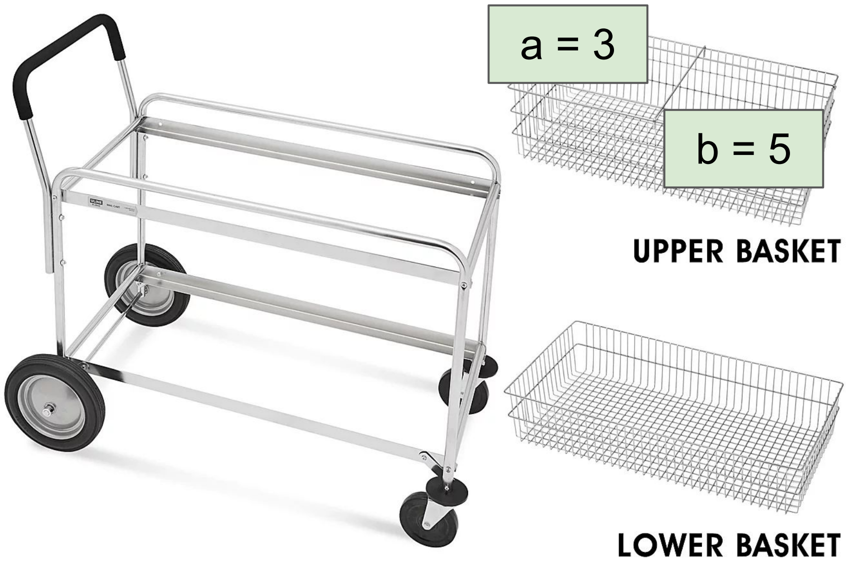
Different data need different sized packages

- E.g., an int in Python requires 4 bytes (32 bits), but a string requires 1-4 bytes per character, plus another 48 bytes of metadata/overhead

Memory addresses

Memory addresses tell us where on the cart (where in RAM) our variable lives

E.g., **int a** has a value of 3, but a memory address of “upper basket, top left corner”, and its box size is 4 bytes

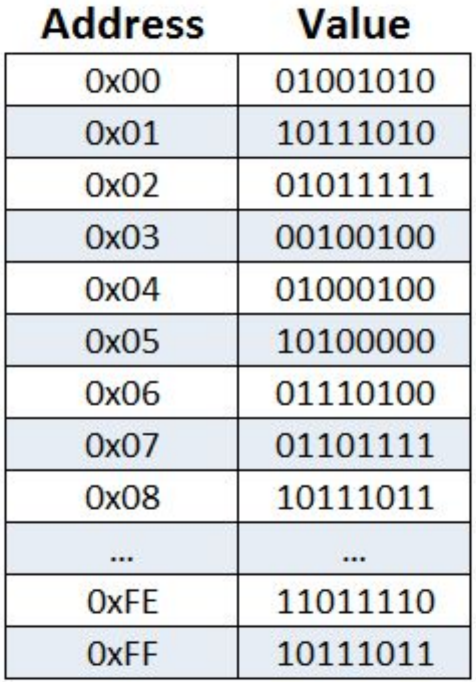


Memory addresses

In reality, addresses are indexed locations on computer hardware

The value at that location is the value of the variable (“3”)

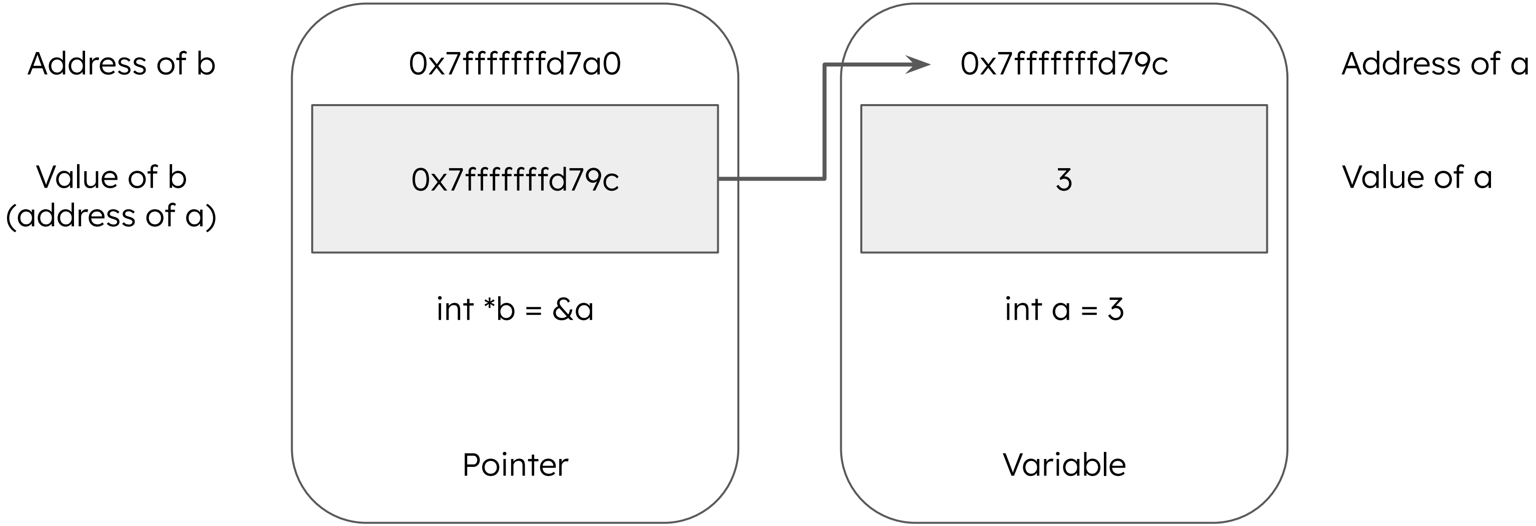
The program understands that when we say “a”, we mean “get me the value that lives at a’s address, which is 3”



Address	Value
0x00	01001010
0x01	10111010
0x02	01011111
0x03	00100100
0x04	01000100
0x05	10100000
0x06	01110100
0x07	01101111
0x08	10111011
...	...
0xFE	11011110
0xFF	10111011

Pointers

Pointers are special ints whose values are memory addresses for other variables, kind of like a scavenger hunt prize that's just another clue



Address of b

0x7fffffffd7a0

Value of b
(address of a)

0x7fffffffd79c

int *b = &a

Pointer

0x7fffffffd79c

Address of a

3

Value of a

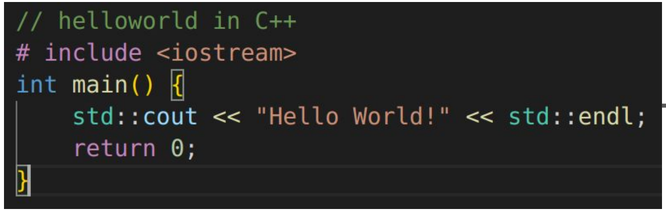
int a = 3

Variable

Getting started in C++

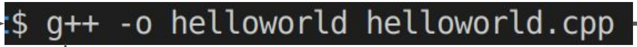
“Hello World” in C++

helloworld.cpp



```cpp
// helloworld in C++
# include <iostream>
int main() {
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

In terminal or from your Dockerfile, compile helloworld.cpp to an executable



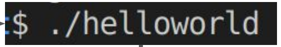
```
$ g++ -o helloworld helloworld.cpp
```

Compiler

Desired executable name

Your script

Run the executable



```
$ ./helloworld
```

Output:
Hello World!

C++ types: pointers vs. references

Remember that in C++, data is statically typed

- int a = 1;
    - “a” is a variable of type integer with value 1

Pointers are variables of type “special integer”

- int* b = &a;
    - “b” is an integer that corresponds to a memory address
    - I think of this as b having type “int*”

C++ types: pointers vs. references

References are aliases for existing variables

- int& c = a;
  - “c” is a reference to the integer “a”
  - I think of this as c having type “int&”, or reference-to-int
  - In most cases c will behave exactly like a

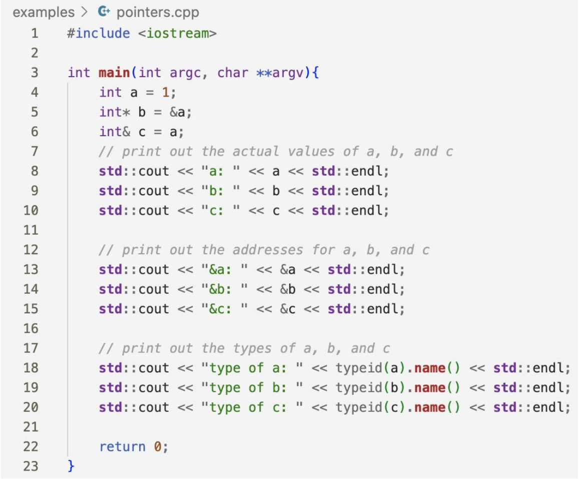
Using pointers and references

On the left side of variable declarations

- These help define the type
- int* to declare a pointer
- int& to declare a reference to an int

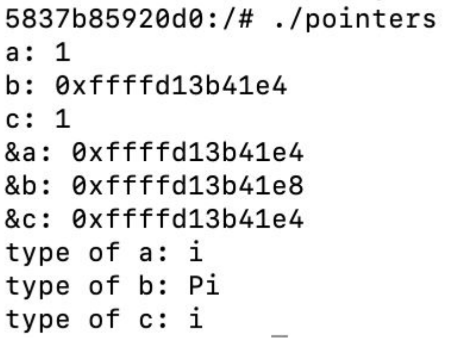
When used with existing variables

- Use “&” to reference an address
- Use “*” to dereference an address



examples > pointers.cpp

```cpp
#include <iostream>

int main(int argc, char **argv){
    int a = 1;
    int* b = &a;
    int& c = a;
    // print out the actual values of a, b, and c
    std::cout << "a: " << a << std::endl;
    std::cout << "b: " << b << std::endl;
    std::cout << "c: " << c << std::endl;

    // print out the addresses for a, b, and c
    std::cout << "&a: " << &a << std::endl;
    std::cout << "&b: " << &b << std::endl;
    std::cout << "&c: " << &c << std::endl;

    // print out the types of a, b, and c
    std::cout << "type of a: " << typeid(a).name() << std::endl;
    std::cout << "type of b: " << typeid(b).name() << std::endl;
    std::cout << "type of c: " << typeid(c).name() << std::endl;

    return 0;
}
```



```
5837b85920d0:/# ./pointers
a: 1
b: 0xffffd13b41e4
c: 1
&a: 0xffffd13b41e4
&b: 0xffffd13b41e8
&c: 0xffffd13b41e4
type of a: i
type of b: Pi
type of c: i
```

C++ types: arrays vs. vectors

- Vectors are like arrays, but they are dynamic
- Vectors can be resized, arrays cannot
- Adding new elements to a vector is slow and dynamic resizing may take up more memory than is needed
  - You should reserve the amount of memory you need when you declare a vector!!!

```
int my_array[3] = {1,2,3}; // d is an array of integers
std::vector<int> my_vector = {1,2,3}; // e is a vector of integers
my_vector.push_back(4); // add 4 to the end of my_vector
my_vector.pop_back(); // remove the last element of my_vector so that it is the same size as my_array
my_vector.reserve(100); // reserve space for 100 integers in my_vector
```

Pointers to arrays, and arrays of pointers

- Pointer to an array
  - int (*pntr_array)[5]; // a pointer to an array of 5 ints
- Array of pointers
  - int *pntr_array[5]; // an array of 5 pointers to integers
- Pointer to a vector
  - std::vector<int>*
- Vector of pointers
  - std::vector<int*>

Arrays are pointers to blocks of memory

- Arrays just point to the start of a memory block
- Array indices are just pointer arithmetic and dereferencing combined
  - a[12] is the same as *(a + 12)
  - &a[3] is the same as a + 3

```
const char *word = “hello”;
word = hello
(word + 1) = ello
word[0] = h
*word = h
word[1] = e
*(word + 1) = e
```

- Large arrays should be dynamically allocated (on the heap)

```
int n = some_large_number;
double * d = new double[n];
```

- Make sure you delete them

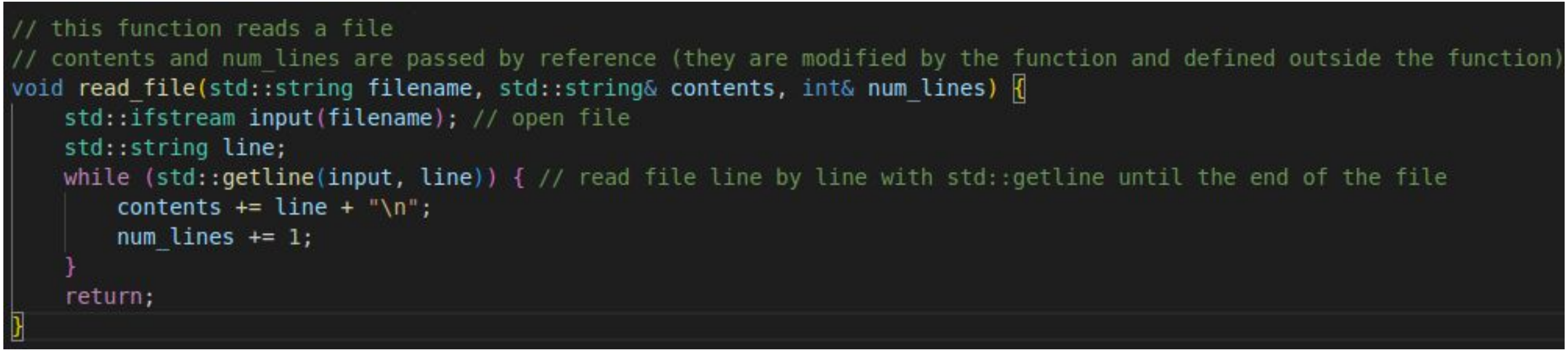
```
delete[] d;
```

Structs are a custom data type in C++

- Structs are like a very simple class
- Used to store data
- Can contain variables of any type (including pointers and other structs)

```
struct my_struct {
        int my_int;
        double my_double;
        std::string my_string;
        std::vector<int> my_vector;
        };
```

Reading Files

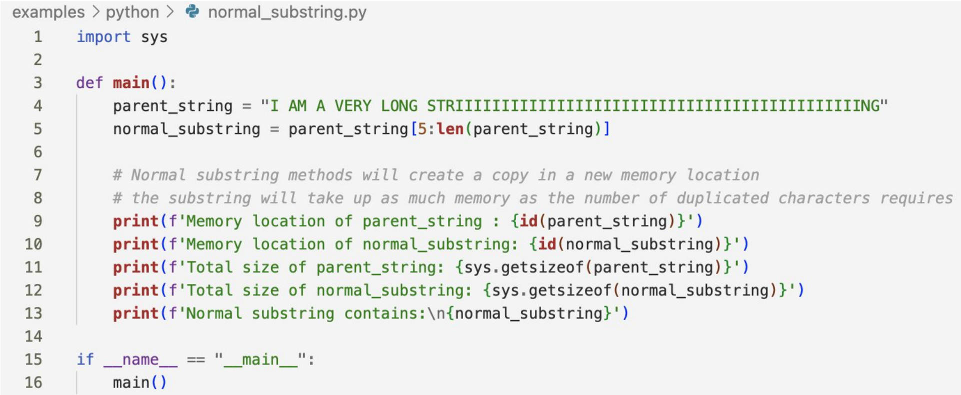


```cpp
// this function reads a file
// contents and num_lines are passed by reference (they are modified by the function and defined outside the function)
void read_file(std::string filename, std::string& contents, int& num_lines) {
    std::ifstream input(filename); // open file
    std::string line;
    while (std::getline(input, line)) { // read file line by line with std::getline until the end of the file
        contents += line + "\n";
        num_lines += 1;
    }
    return;
}
```

Getting around Python

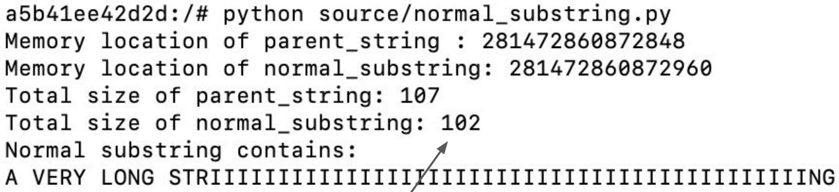
Substrings make new strings

In general, getting the substring of an existing string makes a new string in a new memory location, taking up as much memory as the original string, minus excluded characters



examples > python > normal_substring.py

```python
import sys

def main():
    parent_string = "I AM A VERY LONG STRIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIING"
    normal_substring = parent_string[5:len(parent_string)]

    # Normal substring methods will create a copy in a new memory location
    # the substring will take up as much memory as the number of duplicated characters requires
    print(f'Memory location of parent_string : {id(parent_string)}')
    print(f'Memory location of normal_substring: {id(normal_substring)}')
    print(f'Total size of parent_string: {sys.getsizeof(parent_string)}')
    print(f'Total size of normal_substring: {sys.getsizeof(normal_substring)}')
    print(f'Normal substring contains:\n{normal_substring}')

if __name__ == "__main__":
    main()
```

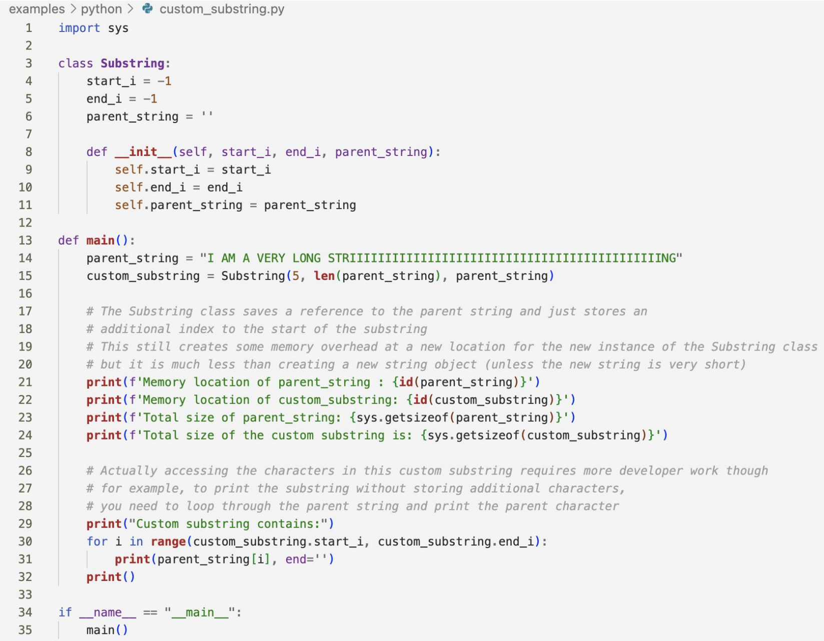


```
a5b41ee42d2d:/# python source/normal_substring.py
Memory location of parent_string : 281472860872848
Memory location of normal_substring: 281472860872960
Total size of parent_string: 107
Total size of normal_substring: 102
Normal substring contains:
A VERY LONG STRIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIING
```

only saved 5 bytes!

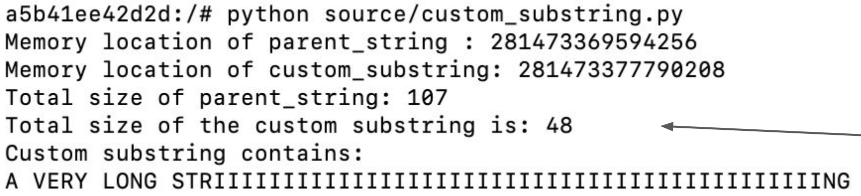
Custom substring class

A way around this is to implement a new “substring” class that holds a reference to a shared “parent” string and a starting index



examples > python > custom_substring.py

```python
import sys

class Substring:
    start_i = -1
    end_i = -1
    parent_string = ''

    def __init__(self, start_i, end_i, parent_string):
        self.start_i = start_i
        self.end_i = end_i
        self.parent_string = parent_string

def main():
    parent_string = "I AM A VERY LONG STRIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIING"
    custom_substring = Substring(5, len(parent_string), parent_string)

    # The Substring class saves a reference to the parent string and just stores an
    # additional index to the start of the substring
    # This still creates some memory overhead at a new location for the new instance of the Substring class
    # but it is much less than creating a new string object (unless the new string is very short)
    print(f'Memory location of parent_string : {id(parent_string)}')
    print(f'Memory location of custom_substring: {id(custom_substring)}')
    print(f'Total size of parent_string: {sys.getsizeof(parent_string)}')
    print(f'Total size of the custom substring is: {sys.getsizeof(custom_substring)}')

    # Actually accessing the characters in this custom substring requires more developer work though
    # for example, to print the substring without storing additional characters,
    # you need to loop through the parent string and print the parent character
    print("Custom substring contains:")
    for i in range(custom_substring.start_i, custom_substring.end_i):
        print(parent_string[i], end='')
    print()

if __name__ == "__main__":
    main()
```



```
a5b41ee42d2d:/# python source/custom_substring.py
Memory location of parent_string : 281473369594256
Memory location of custom_substring: 281473377790208
Total size of parent_string: 107
Total size of the custom substring is: 48
Custom substring contains:
A VERY LONG STRIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIING
```

huge savings!

Other Python tips

Some libraries are can help workflow efficiency

- Numpy for numerical data and matrix math
- Pandas for managing tabular data
- Cython for compiling Python down to C

If you have other Python questions, feel free to Slack me!

Questions?

About the discussion topic or the homework!

Image references

[Mail room]

[Pushed mail cart]

[Cart upper lower]

[Memory addresses]