

# 03: C Interfaces

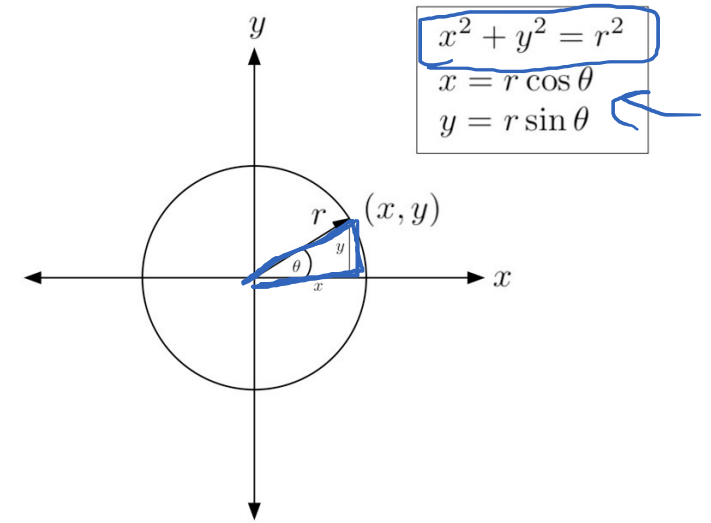
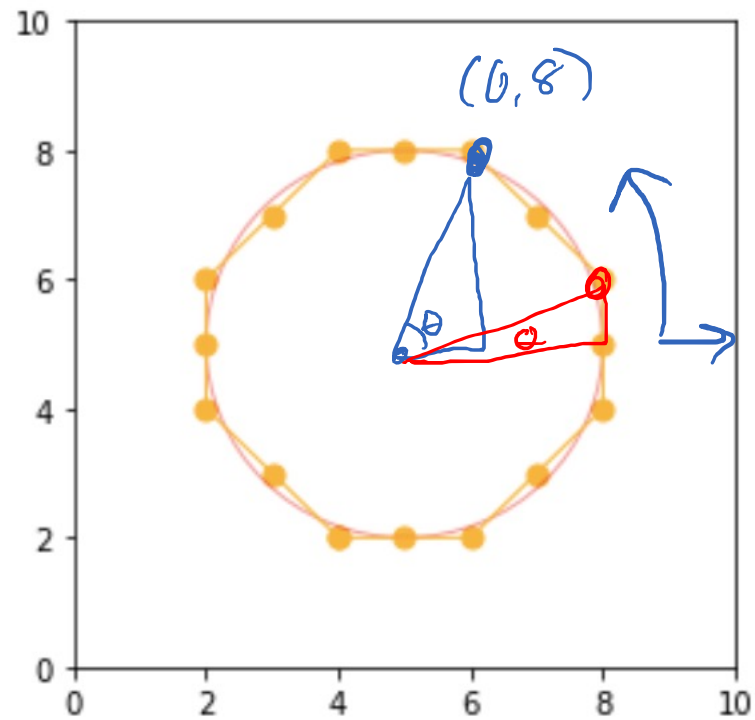
Engr 315: Hardware / Software Codesign  
Andrew Lukefahr  
*Indiana University*



# Announcements

- Slack – See Website
- Office Hours – See Website / Syllabus
- P1: Due Friday
- P2: Out now!
  - (New Project, Could be some bumps)

# Project 1: Circles



# Project 1: Circles

- This is optimized already:

```
def computeTheta(self, x,y, x_centre, y_centre):  
    return math.atan2(x-x_centre, y-y_centre)
```

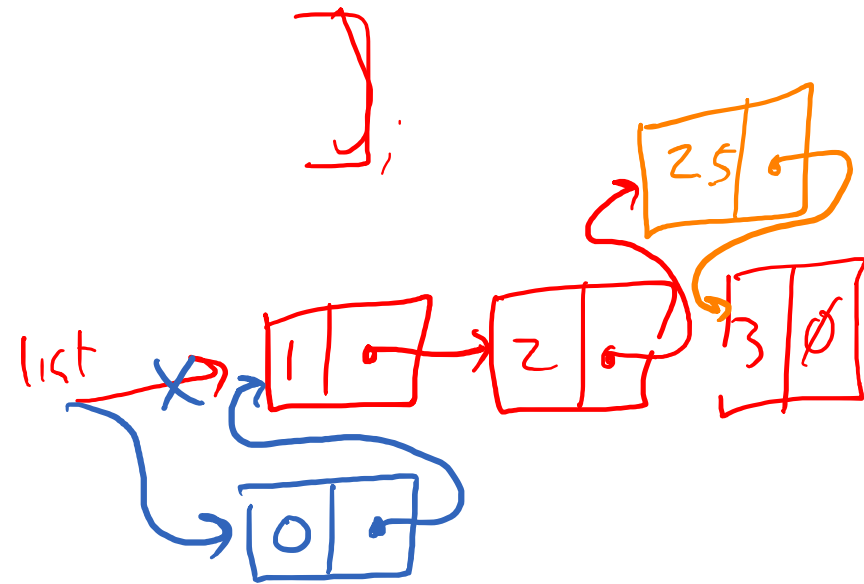
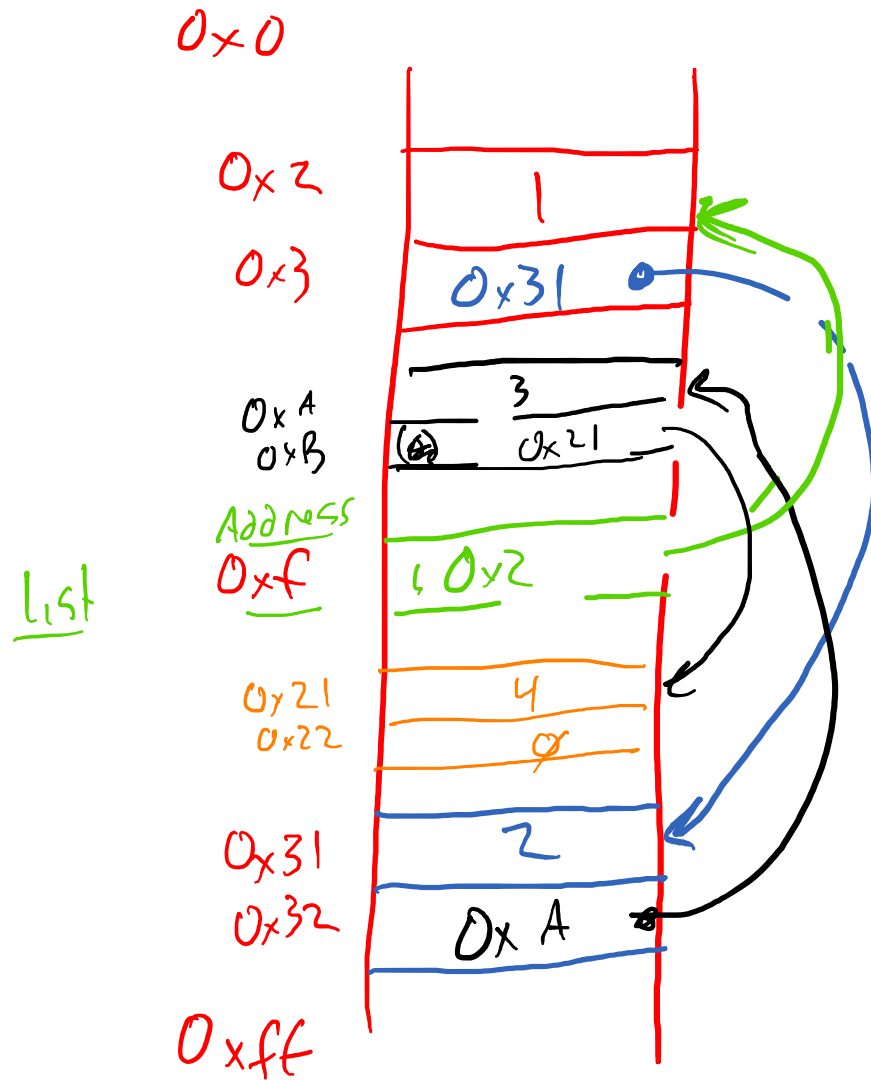
- You aren't going to accelerate it. Don't try.
- Figure out how to call it less.

# Project 2: Accelerate Exp. Moving Avg.

- Need Pynq board for P2.
- Hope to have those ready by Wednesday.

# P2 Demo Time

linked list = [ 1, 2, 3, ]



# Array vs. Linked List: Random Access

```
1 def traverse( thing, times):
2     idx = 0
3     for i in range(times):
4         idx = thing[idx]
5
6 random.seed(1)
7 sz = 1000000
8 nums = [x for x in range(sz)]
9 random.shuffle(nums)
10 random.shuffle(nums)
11 lst = collections.deque(nums)
12 arr = np.array(nums)
13 trips = 1000
14
15 start_time = time.time()
16 traverse(lst, trips)
17 end_time = time.time()
18 print("True List: %f seconds" % (end_time - start_time))
19
20 start_time = time.time()
21 traverse(arr, trips)
22 end_time = time.time()
23 print("Array: %f seconds" % (end_time - start_time))
24
25 start_time = time.time()
26 traverse(nums, trips)
27 end_time = time.time()
28 print("Python List: %f seconds" % (end_time - start_time))
```

True List: 0.037878 seconds

Array: 0.000312 seconds

Python List: 0.000410 seconds



# Array vs. Linked List: Sequential Insert

Insert at: 0

True List: 0.000085 seconds

Array: 0.335853 seconds

Python List: 0.115629 seconds

Insert at: 750000  $\sim 3/4$

True List: 0.054327 seconds

Array: 0.336377 seconds

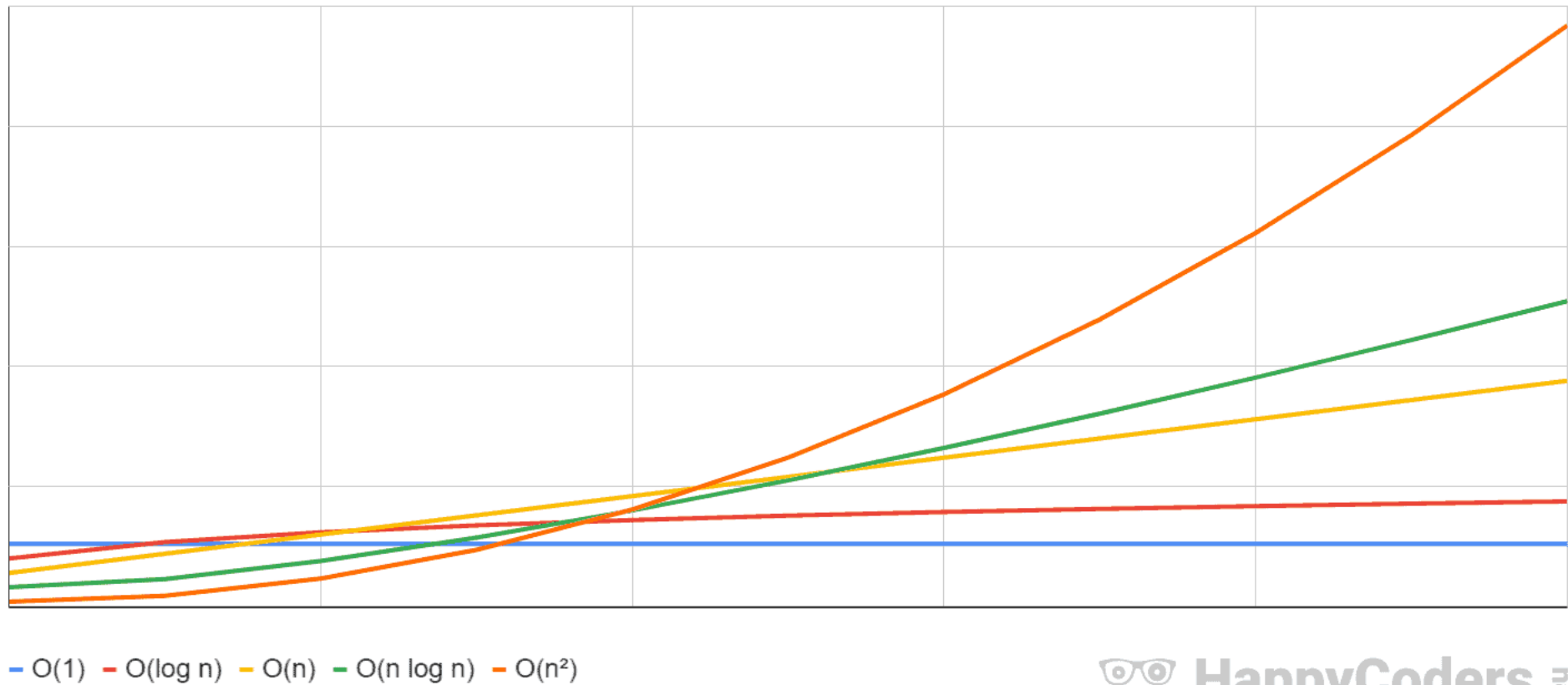
Python List: 0.022257 seconds

# Big O Complexity

- Computational time complexity describes the change in the runtime of an algorithm, depending on the change in the input data's size.
- "How much does an algorithm's performance change when the amount of input data changes?"

# O() Complexities

Comparing the complexity classes  $O(1)$ ,  $O(\log n)$ ,  $O(n)$ ,  $O(n \log n)$ ,  $O(n^2)$



Material taken from: <https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/>

# Conclusion #3: Think about your data structure!

- How will you be accessing your data?
  - Randomly? Sequentially?
- How will you up updating your data?
- Pick a data structure to minimize overheads for your access patterns

# Find: The needle in the haystack.

```
1 def find_ignore_case( needle, haystack):
2     results = []
3     for hi in range(len(haystack)):
4         match = True
5         for ni in range(len(needle)):
6             h = haystack[hi + ni].lower()
7             n = needle[ni].lower()
8             if h != n:
9                 match=False
10                break
11        if match:
12            results.append(hi)
13    return results
14
```

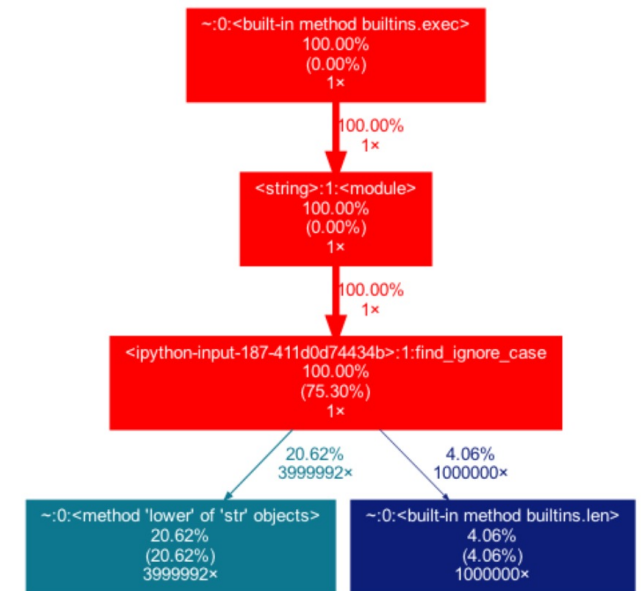
```
28 sz=20
29 haystack = random_str(sz)
30 needle = haystack[int(sz/2):int(sz/2)+2]
31 results = find_ignore_case(needle, haystack)
32
33 print (needle)
34 print (haystack)
35 print (results)
36
```

```
sk
eiPPzDAnWiskaumnqYpl
[10]
```

# Find: The needle in the haystack.

```
1 def find_ignore_case( needle, haystack):
2     results = []
3     for hi in range(len(haystack)-len(needle)):
4         match = True
5         for ni in range(len(needle)):
6             h = haystack[hi + ni].lower()
7             n = needle[ni].lower()
8             if h != n:
9                 match=False
10        if match:
11            results.append(hi)
12    return results
13
14 random.seed(1)
15 sz=1000000
16 haystack = random_str(sz)
17 needle = haystack[int(sz/2):int(sz/2)+2]
18
19 start_time = time.time()
20 results = find_ignore_case(needle, haystack)
21 end_time = time.time()
22 print("True List: %f seconds" % (end_time - start_time))
```

True List: 1.109001 seconds



• No libraries!

# Find: The needle in the haystack.

```
def find_ignore_case( needle, haystack):
    results = []
    for hi in range(len(haystack)-len(needle)):
        match = True
        for ni in range(len(needle)):
            h = haystack[hi + ni].lower()
            n = needle[ni].lower()
            if h != n:
                match=False
        if match:
            results.append(hi)
    return results
```

```
def find_ignore_case2( needle, haystack):
    results = []
    needle = needle.lower() # new
    haystack = haystack.lower() # new
    for hi in range(len(haystack)-len(needle)):
        match = True
        if haystack[hi] == needle[0]:
            for ni in range(len(needle)-1):
                h = haystack[hi + ni]#.lower()
                n = needle[ni]#.lower()
                if h != n:
                    match=False

            if match:
                results.append(hi)
                break # new
    return results
```

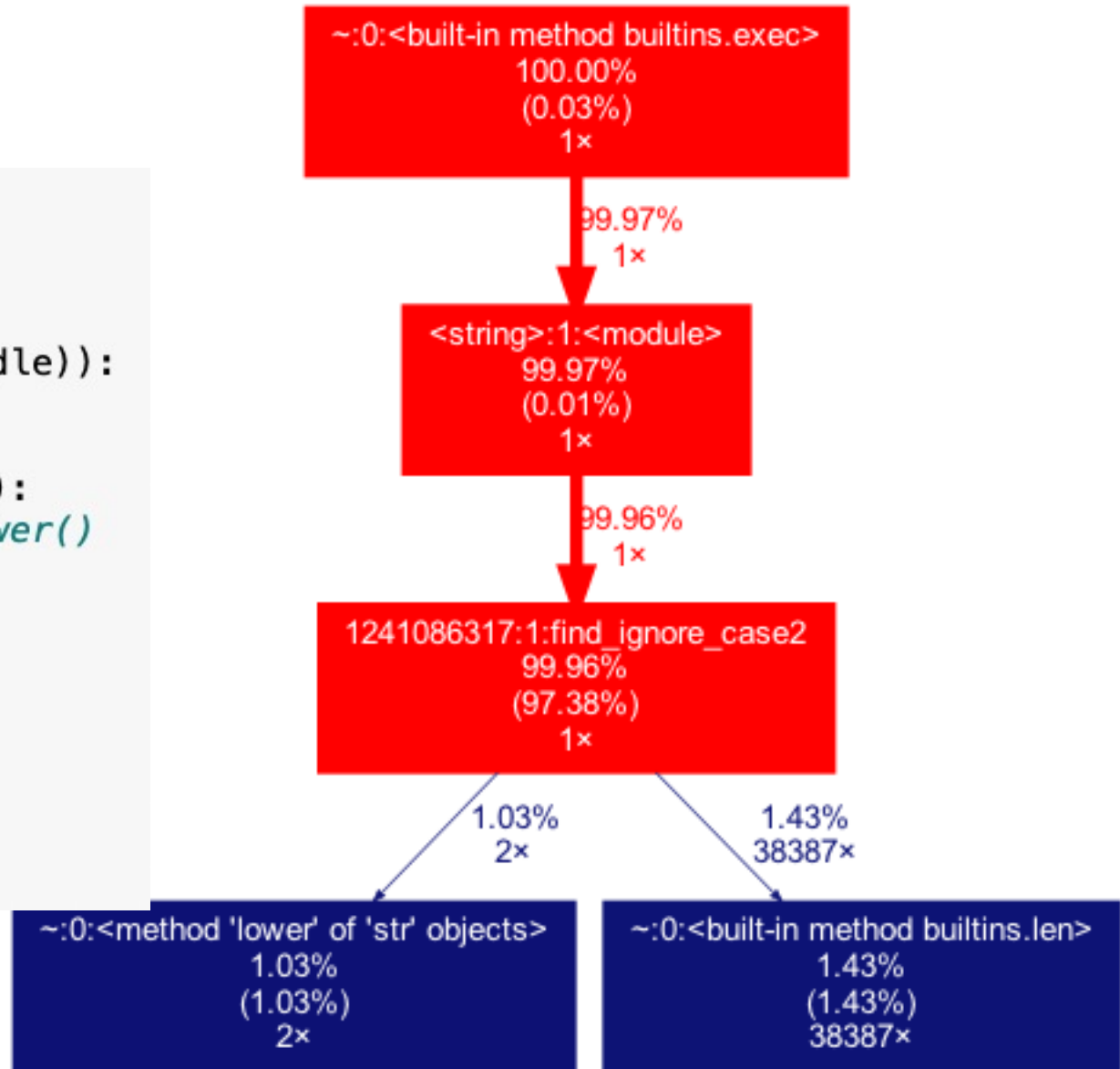
Find: 0.917540 seconds

Find2: 0.440155 seconds

# Anything else?

```
def find_ignore_case2( needle, haystack):
    results = []
    needle = needle.lower() # new
    haystack = haystack.lower() # new
    for hi in range(len(haystack)-len(needle)):
        match = True
        if haystack[hi] == needle[0]:
            for ni in range(len(needle)-1):
                h = haystack[hi + ni]#.lower()
                n = needle[ni]#.lower()
                if h != n:
                    match=False

            if match:
                results.append(hi)
                break # new
    return results
```





# Anything else?

```
def find_ignore_case3( needle, haystack):  
    results = []  
    needle = needle.lower() # new  
    haystack = haystack.lower() # new  
    r = range(len(needle)-1) # new  
  
    for hi in range(len(haystack)-len(needle)):  
        match = True  
  
        if haystack[hi] == needle[0]:  
            for ni in r: # update  
                h = haystack[hi + ni].lower()  
                n = needle[ni].lower()  
                if h != n:  
                    match=False  
  
            if match:  
                results.append(hi)  
                break # new  
    return results
```

Find: 0.370030 seconds

Find2: 0.057817 seconds

Find3: 0.053763 seconds

[New Mac Times]

```

def find_ignore_case4( needle, haystack):
    results = []
    needle = needle.lower() # new
    haystack = haystack.lower() # new
    r = range(len(needle)-1) # new

    for hi in range(len(haystack)-len(needle)):
        #match = False

        if haystack[hi] == needle[0]:
            for ni in r: # update
                h = haystack[hi + ni].lower()
                n = needle[ni].lower()
                if h == n: # new
                    #match=False
                    results.append(hi) # new
                    break # new
            #if match:
            #results.append(hi)
    return results

```

```

Find: 0.259516 seconds
Find2: 0.057128 seconds
Find3: 0.053053 seconds
Find4: 0.048197 seconds

```

Q: How is this so much faster?

```
def find_ignore_case5( needle, haystack):  
    return [haystack.find(needle)]
```

```
Find: 0.259516 seconds  
Find2: 0.057128 seconds  
Find3: 0.053053 seconds  
Find4: 0.048197 seconds  
Find5: 0.000172 seconds
```

A: It built into Python. So it runs at C speeds!

# Really faster?

```
def find_ignore_case5( needle, haystack):  
    return [haystack.find(needle)]
```

Numpy is written in C/Assembly. It's faster.

```
import numpy
def find_ignore_case6(needle, haystack):
    return np.where(haystack==needle)
```

Find: 0.270210 seconds  
Find2: 0.061821 seconds  
Find3: 0.054265 seconds  
Find4: 0.051191 seconds  
Find5: 0.000265 seconds  
Find6: 0.000052 seconds

# Conclusions

1. Optimize algorithm if possible
2. Function calls are not free!
3. Preallocation (Bulk Allocation) is *usually* faster
4. Think about your data structure!
5. Use optimized libraries if possible

# Popcount

- Count the number of binary 1's in a number
- 0**1**000**1**00**1**0**1**0000**1**00**1**0000**1**00000000
- 7 total 1's

# Popcount

```
def popcount(num):  
    return bin(num).count('1')
```

5 → "0101"

```
value: 0 bin: 0b0 popcount: 0  
value: 1 bin: 0b1 popcount: 1  
value: 2 bin: 0b10 popcount: 1  
value: 3 bin: 0b11 popcount: 2  
value: 4 bin: 0b100 popcount: 1  
value: 5 bin: 0b101 popcount: 2  
value: 6 bin: 0b110 popcount: 2  
value: 7 bin: 0b111 popcount: 3  
value: 8 bin: 0b1000 popcount: 1  
value: 9 bin: 0b1001 popcount: 2
```



# popcount (slower, but no external calls)

```
def popcount2 (num) :  
    w = 0  
    while (num) :  
        w += 1  
        num &= num - 1  
    return w
```

# popcount (slower, but no external calls)

```
def popcount2(num):
```

```
    w = 0
```

```
    → while (num):
```

```
        w += 1
```

```
        num &= num - 1
```

```
    return w
```

```
→ def popcount2_all(buf):
```

```
    return sum(map(bitcount2, buf))
```

↓  
→ 1011 = 11

✱ 1010(-1) + |

1010

✱ 1001 + |

1000

✱ 0111 + + |

0000 -

3

`Popcount_all` is a helper function to run larger blocks of inputs

```
def popcount_all(buf):  
    return sum(map(popcount,buf))
```

```
def popcount2_all(buf):  
    return sum(map(popcount2,buf))
```

# Big Bitcount

```
np.random.seed(1)
buf = np.random.randint(0, 1E9, int(1E6))

start_time = time.time()
sum_1s = popcount_all(buf)
end_time = time.time()
print("popcount: %f seconds (w/lib)"
      % (end_time - start_time))

start_time = time.time()
sum_1s = popcount2_all(buf)
end_time = time.time()
print("popcount2: %f seconds (w/o lib)"
      % (end_time - start_time))
```

```
popcount: 0.307169 seconds (w/lib)
popcount2: 1.853192 seconds (w/o lib)
```

# How did the library go so much faster?

- Python called C.
- The computations happened in C. It's faster.
- Can we do that?

Let's find out.

# Popcount in Python vs. C

## Python

```
def popcount2(num):  
  
    w = 0  
    while (num):  
        w += 1  
        num &= num - 1  
  
    return w
```

## C

```
int popcount(uint64_t num)  
{  
    int w=0;  
    while (num) {  
        w +=1;  
        num &= (num -1);  
    }  
    return w;  
}
```

# Popcount test?

```
#include <stdio.h>
#include "popcount.h"

int main()
{
    int res;
    for (int i = 0; i < 20; ++i){
        res = popcount(i);
        printf ("i:%d i:0x%x res: %d\n", i, i, res);
    }
    return 0;
}
```

```
i:0 i:0x0 res: 0
i:1 i:0x1 res: 1
i:2 i:0x2 res: 1
i:3 i:0x3 res: 2
i:4 i:0x4 res: 1
i:5 i:0x5 res: 2
i:6 i:0x6 res: 2
i:7 i:0x7 res: 3
i:8 i:0x8 res: 1
i:9 i:0x9 res: 2
i:10 i:0xa res: 2
i:11 i:0xb res: 3
i:12 i:0xc res: 2
i:13 i:0xd res: 3
i:14 i:0xe res: 3
i:15 i:0xf res: 4
i:16 i:0x10 res: 1
i:17 i:0x11 res: 2
i:18 i:0x12 res: 2
i:19 i:0x13 res: 3
```

# Let's see if we can wrap C popcount with Python

- <https://realpython.com/build-python-c-extension-module/#packaging-your-python-c-extension-module>
- <https://docs.python.org/3/extending/extending.html>



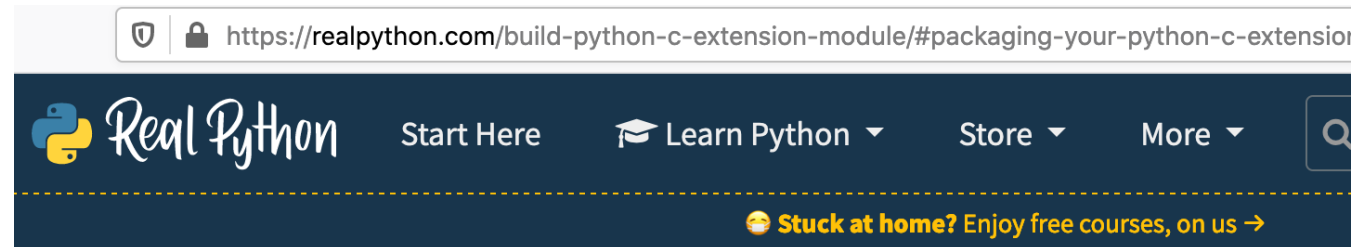
# Step 1: RTFM



```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return PyLong_FromLong(sts);
}
```

## Step 2: RTFM 2



A minimal `setup.py` file for your module should look like this:

Python

```
from distutils.core import setup, Extension

def main():
    setup(name="fputs",
          version="1.0.0",
          description="Python interface for the fputs C library function",
          author="<your name>",
          author_email="your_email@gmail.com",
          ext_modules=[Extension("fputs", ["fputsmodule.c"])]])

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

```

20 static PyObject *
21 cPopcount(PyObject *self, PyObject *args)
22 {
23     uint64_t num;
24
25     if (!PyArg_ParseTuple(args, "l", &num))
26         return NULL;
27
28     //popcount!!!
29     uint64_t res = popcount(num);
30
31     return PyLong_FromLong(res);
32 }

```

```

int popcount(uint64_t num)
{
    int w=0;
    while (num) {
        w +=1;
        num &= (num -1);
    }
    return w;
}

```

```

import cPopcount
cPopcount.cPopcount(0xffff)

```

```

np.random.seed(1)
buf = np.random.randint(0,1E9,int(1E6))
buf = buf.tolist()

start_time = time.time()
sum_1s = popcount_all(buf)
end_time = time.time()
print("popcount: %f seconds (w/calls)"
      % (end_time - start_time))

start_time = time.time()
sum_1s = popcount2_all(buf)
end_time = time.time()
print("popcount2: %f seconds (w/o calls)"
      % (end_time - start_time))

start_time = time.time()
sum_1s = sum(map(cPopcount.cPopcount,buf))
end_time = time.time()
print("c_popcount: %f seconds (64-bits in C)"
      % (end_time - start_time))

```

```

popcount: 0.261108 seconds (w/calls)
popcount2: 0.881429 seconds (w/o calls)
c_popcount: 0.027510 seconds (64-bits in C)

```

# Can we do cPopcount\_all in C?

- Send an entire list to C?

```

static PyObject *
cPopcount_all(PyObject *self, PyObject *args)
{
    PyObject *obj;
    int64_t res = 0;

    //parse the list argument
    if (!PyArg_ParseTuple(args, "O", &obj)) {
        return NULL;
    }

    //hope it's iterable
    PyObject *iter = PyObject_GetIter(obj);
    if (!iter) {
        return NULL; // error not iterator
    }

    //loop over all elements in list
    while (1) {
        PyObject *next = PyIter_Next(iter);

        if (!next) {
            // nothing left in the iterator
            break;
        }
    }
}

```

```

    // convert to int64_t
    int64_t num = 0;
    if (PyLong_Check(next)) {
        num = PyLong_AsLong(next);
    } else {
        printf("unsupported type\n");
        return NULL;
    }

    //now do popcount!
    res += popcount(num); // do something with foo

    /* release reference when done */
    Py_DECREF(next);
}
Py_DECREF(iter);

return PyLong_FromLong(res);
}

```

# Two ways to handle lists:

- Iterators (previous slide)
- <https://stackoverflow.com/questions/22458298/extending-python-with-c-pass-a-list-to-pyarg-parsetuple>
- Array indices (not shown)
- <https://stackoverflow.com/questions/39063112/passing-a-python-list-to-c-function-using-the-python-c-api>

```
start_time = time.time()
sum_1s = sum(map(cPopcount.cPopcount,buf))
end_time = time.time()
print("c_popcount: %f seconds (64-bits in C)"
      % (end_time - start_time))

start_time = time.time()
sum_1s = cPopcount.cPopcount_all(buf)
end_time = time.time()
print("c_popcount: %f seconds (List in C)"
      % (end_time - start_time))
```

```
popcount: 0.261108 seconds (w/calls)
popcount2: 0.881429 seconds (w/o calls)
c_popcount: 0.027510 seconds (64-bits in C)
c_popcount: 0.007329 seconds (List in C)
```



Same algorithm. C vs. Python.

```
popcount: 0.261108 seconds (w/calls)
popcount2: 0.881429 seconds (w/o calls)
c_popcount: 0.027510 seconds (64-bits in C)
c_popcount: 0.007329 seconds (List in C)
```

When performance matters, use C.  
When it doesn't, use Python.

# 03: C Interfaces

Engr 315: Hardware / Software Codesign  
Andrew Lukefahr  
*Indiana University*

