02: C Interfaces

test

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University

Course Website

engr315.github.io

Write that down!

E315's autograder:

- Use this:
- https://autograder.sice.indiana.edu

- Not this:
- https://autograder.luddy.indiana.edu

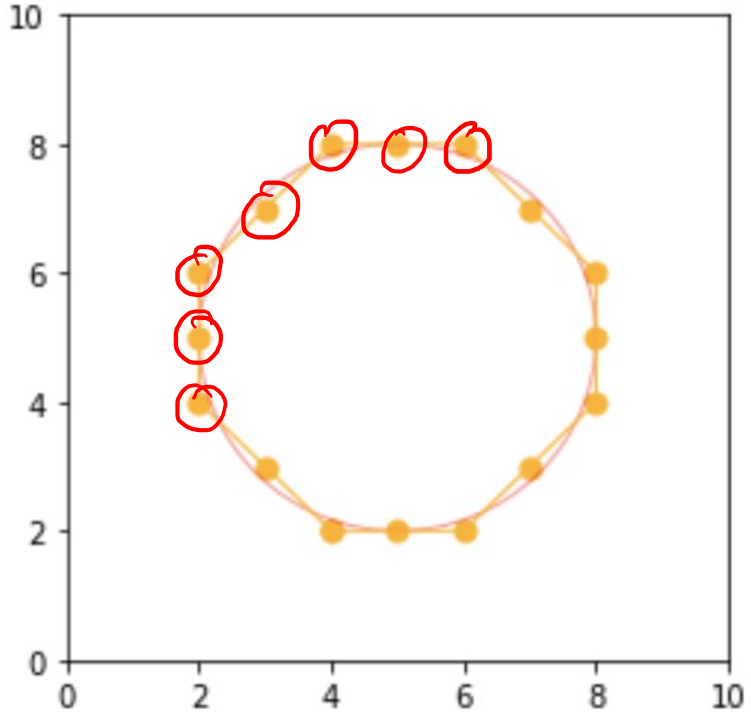
It's a long story…

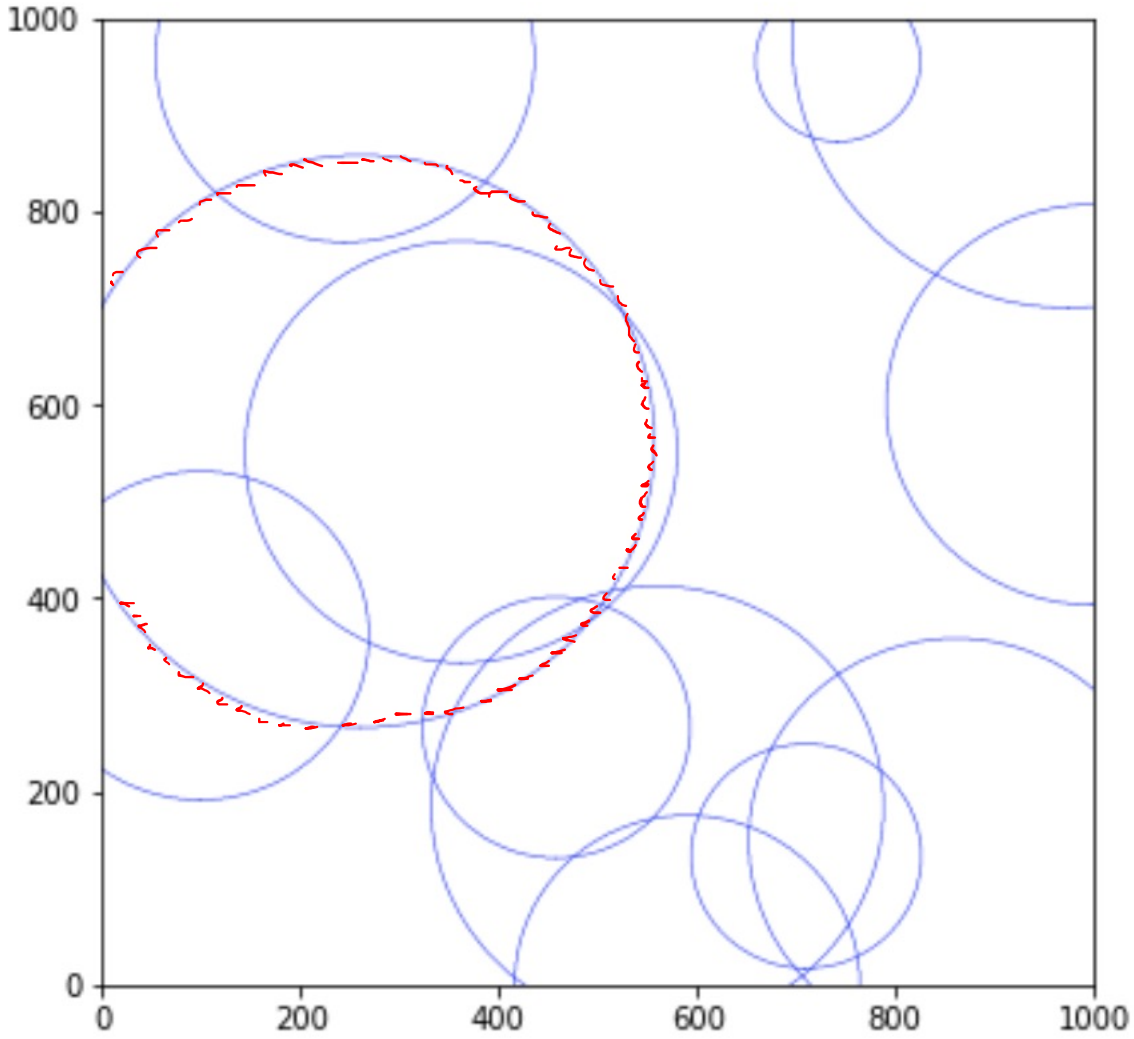
Announcements

- Slack
  - Contact Joe josbella@iu.edu if questions
- Office Hours:
  - Me: 4-5pm T/H
  - Mike: M 5-7pm
    F ~~10-12~~
    10-12am

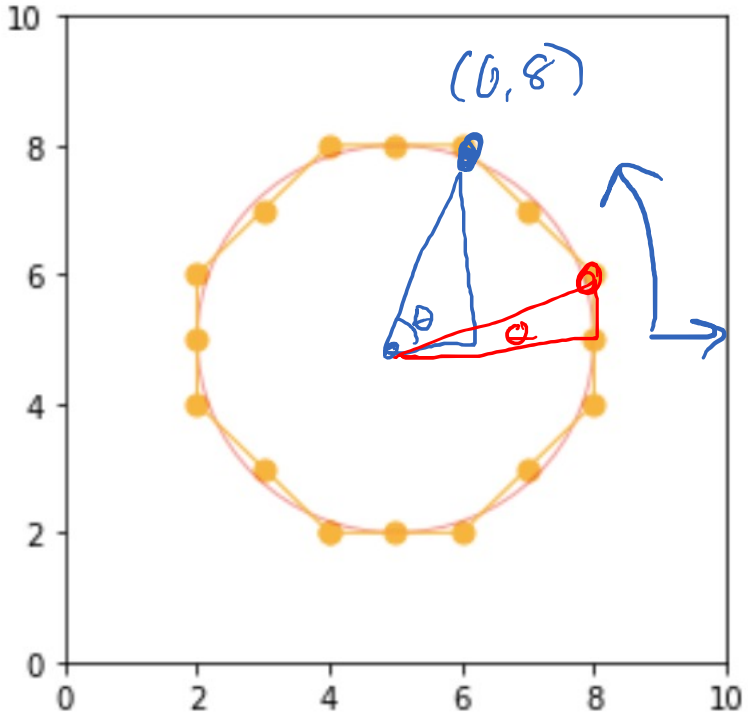
link in Canvas Chat

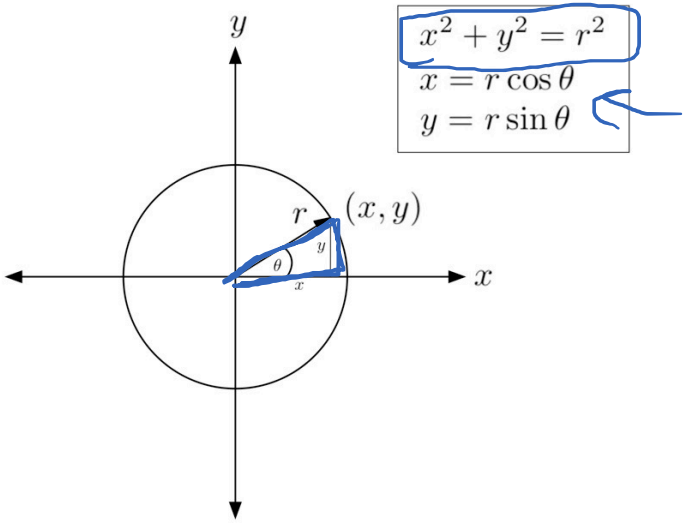
Project 1: Circles





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 1: “Bonus”

- Bonus: 0.119 seconds (my original time)
- Better Bonus: 0.011 seconds (Mike’s time)
- Better Better Bonus: 0.007 seconds (best time last year)

Good luck!

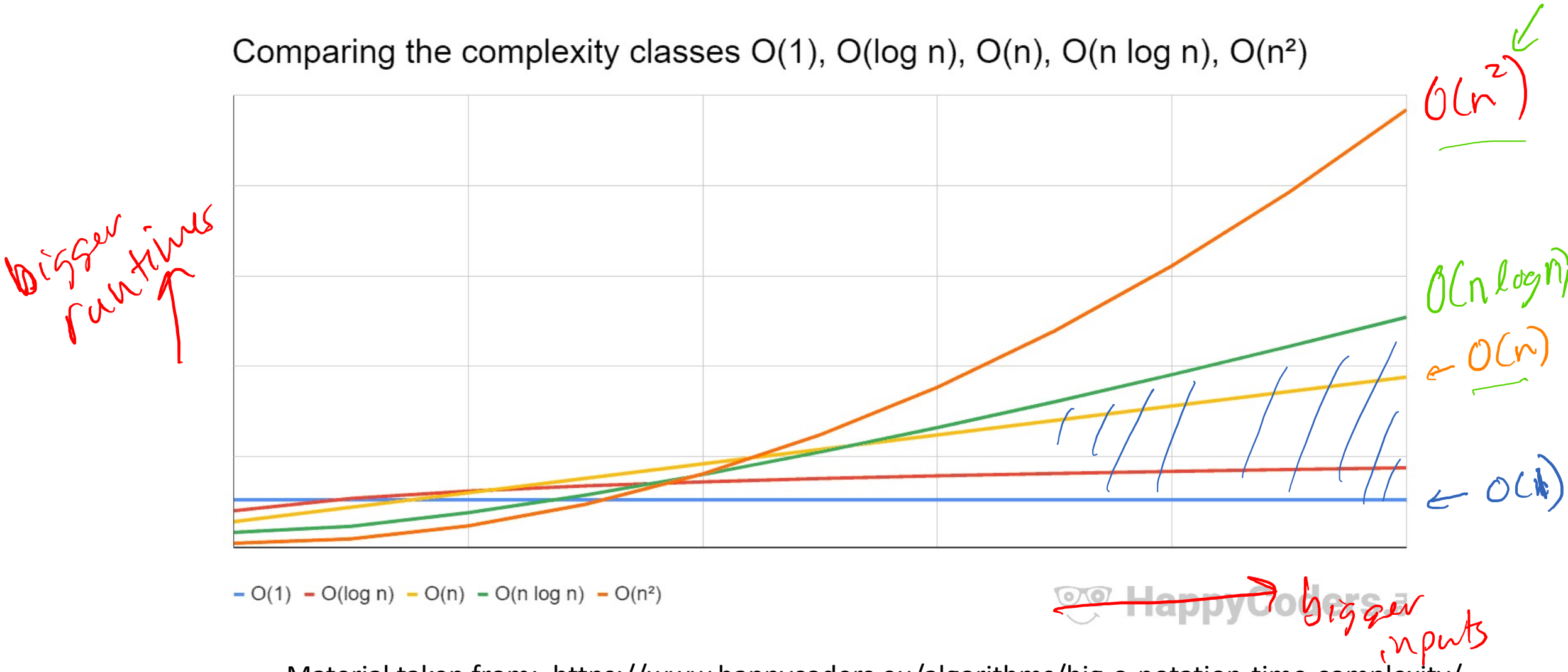
Project 2: EMA

- Need Pynq board for P2.
- Can take one when you are ready.
- Setup is involved. It’s most of the project.

Last Time: Big O Complexity

- "How much does an algorithm’s performance change when the amount of input data changes?"

O() Complexities



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

Last Time: The needle in the haystack.

```python
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
```

```python
def find_ignore_case3( needle, haystack):
    results = []
    needle = needle.lower() # new
    haystack = haystack.lower() # new
    r = range(len(needle)) # 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
                    break # new
            if match:
                results.append(hi)
    return results
```

```
Find: 0.370030 seconds
Find2: 0.057817 seconds
Find3: 0.053763 seconds
```

↑ ~8× faster

0.37 / 0.05

Using built-in libraries is usually faster…

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

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

Numpy is written in C. 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
- 01000100101000010010000100000000
- 7 total 1’s

Popcount

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

```
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
```

$x - y = x + (-y) = x + (\sim y + 1)$

popcount (slower, but no external calls)

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

5 → "0101"

4 → 0100

3 → 0011

0101 w=1

& 0100

0100 w=2

~4 = 0100

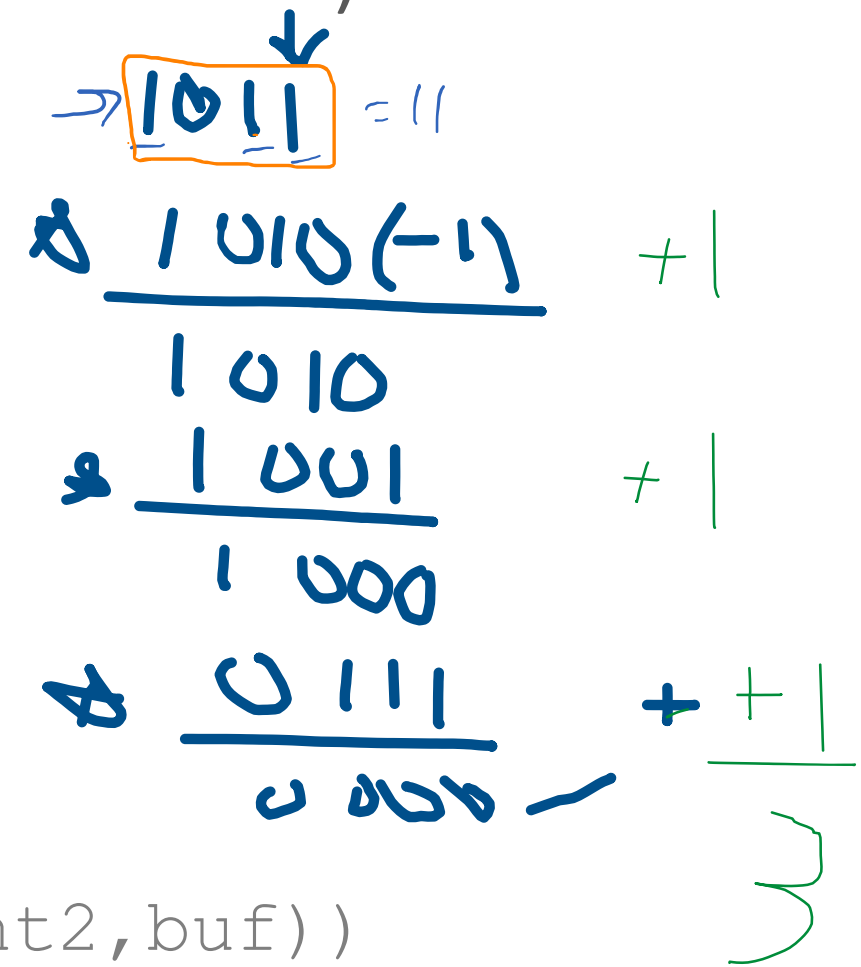
& 0011

0000 ⇒

w=2

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))
```



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

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

popcount

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/libs)"
      % (end_time - start_time))

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

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

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

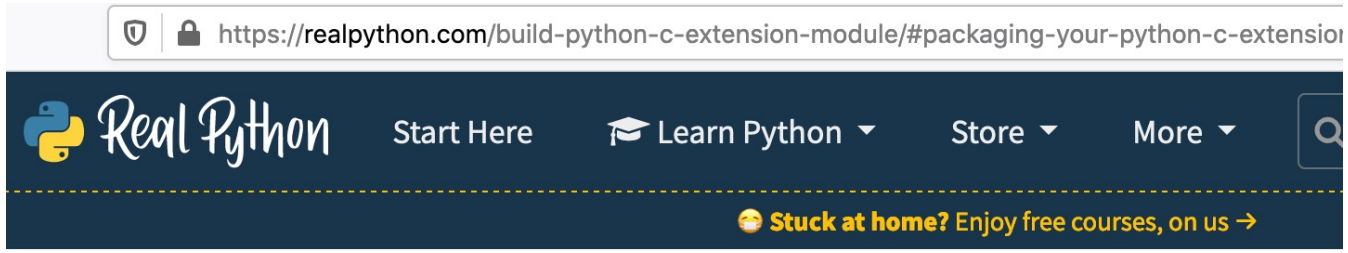
Step 1: RTFM

https://docs.python.org/3/extending/extending.html

```c
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

```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()
```

```c
static PyObject *
cPopcount(PyObject *self, PyObject *args)
{
    uint64_t num;

    if (!PyArg_ParseTuple(args, "l", &num))
        return NULL;

    //popcount!!!
    uint64_t res = popcount(num);

    return PyLong_FromLong(res);
}
```

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

```
import cPopcount
cPopcount.cPopcount(0xffff)
```

```
16
```

```
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)
```

```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 iteratable
    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;
        }
```

```c
        // conver 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);
}
```

```
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.

02: C Interfaces

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University