# Correctness, Style, & Speed

Ruth Anderson

UW CSE 160

Autumn 2022

# Correctness

- Correctness is the most important property of a program!

- We've talked about:
  - Testing
  - Debugging

# Style

- Programs are read by humans
  - Good style is important so humans can understand what you are doing
    - to modify your code and re-use it
    - to debug your code
  - Sometimes this human is you
  - Sometimes it is another person
  - Sometimes it is you a year later

# Style

- We will be grading on style in all remaining homeworks
- Things that were -0 will now actually take off points

- CSE 160 Style guide:

https://courses.cs.washington.edu/courses/cse160/22au/computing/style_guide.html

- Flake8 – helps enforce good style

# Good Names vs. Line Length

- Using good names for variables and functions can make lines long! We want lines =< 80 characters!
- Make use of Python's implicit line joining inside parentheses, brackets and braces. If necessary, you can add an extra pair of parentheses around an expression.

```
foo_bar(self, width, height,
        page_size, grid_size)

if (width == 0 and height == 0 and
    color == 'red' and emphasis == 3):
```

- Can also use \ (as we have done in test code)

http://google.github.io/styleguide/pyguide.html#32-line-length

# Helper functions

- Writing functions allows us to re-use code
  - Avoid duplication – only one place to fix/modify
- Calling helper functions
  - Keeps our functions shorter

# Speed

- Correctness is more important than speed

- Computer time is much cheaper than human time
- The cost of your program depends on:
  - Time to write and verify it
    - High cost: salaries
  - Time to run it
    - Low cost: electricity
- An inefficient program may give you results faster

# Sometimes, speed does matter

- Programs that need to run in real time
  - e.g. will my car crash into the car in front of me?
- Very large datasets
  - Even inefficient algorithms usually run quickly enough on a small dataset
  - Example large data set:

    Google:

    67 billion pages indexed (2014)

    5.7 billion searches per day (2014)

    Number of pages searched per day??

# Program Performance

We'll discuss two things a programmer can do to improve program performance:

- A. Good Coding Practices
- B. Good Algorithm Choice

# Good Coding Practices (1)

- Minimize amount of work inside of loops

```
y = 500
for i in range(n):
    z = expensive_function()
    x = 5.0 * y / 2.0 + z
    lst.append(x + i)
```

Move computations that WILL NOT CHANGE outside/above the loop whenever possible.

# Good Coding Practices (2)

- Minimize amount of work inside of loops

```
for i in friends_of_friends(user):
    for j in friends_of_friends(user):
    # do stuff with i and j
```

Move computations that WILL NOT CHANGE outside/above the loop whenever possible.

# Good Coding Practices (3)

- Avoid iterating over data multiple times when possible

```
for base in nucleotides:
    if base == 'A':
        # code here

for base in nucleotides:
    if base == 'C':
        # code here

for base in nucleotides:
    if base == 'T':
        # code here

for base in nucleotides:
    if base == 'G':
        # code here
```

```
for base in nucleotides:
    if base == 'A':
        # code here

    elif base == 'C':
        # code here

    elif base == 'T':
        # code here

    elif base == 'G':
        # code here
```

Even without the loop, it is more efficient to use the if elif elif than multiple if statements (Potentially fewer cases will be checked with the elif option vs. the if option where all four options will always be checked.)

# Good Coding Practices (4)

- Expensive operations:
  - Reading files
  - Writing files
  - Printing to the screen
- Try to open the file once and read in all the data you need into a data structure.
- Accessing the data structure will be MUCH faster than reading the file a second time.

# Testing and Developing your Program

- Test your program on a SMALL input file.
  - This will allow you to calculate expected results by hand to check for correctness
  - But it can also make your development process easier if you have to wait a shorter time for your program to run

# B. Good Algorithm Choice

- Good choice of algorithm can have a much bigger impact on performance than the good coding practices mentioned.
- However good coding practices can be applied fairly easily
- Trying to come up with a better algorithm can be a (fun!) challenge
- Remember:
  **Correctness** is more important than **speed!!**

# How should we compare the speed of two algorithms?

We are trying to pick the best algorithm to sort integers.

- I say my algorithm runs in 5 seconds
- My friend says their algorithm runs in 4 seconds

What questions do you have for us?

# A Better Way to Compare Two Algorithms

- Hardware?
  - Count number of “operations” something independent of speed of processor
- Properties of data set? (e.g. almost sorted, all one value, reverse sorted order)
  - Pick the worst possible data set: gives you an upper bound on how long the algorithm will take
  - Also it can be hard to decide on what is and “average” data set
- Size of data set?
  - Describe running time of algorithm as a function of data set size

# How fast is an algorithm?

- We describe running time of algorithm as a **function of the data set size (n)**

# Asymptotic Analysis

- Comparing “Orders of Growth”
- This approach works <u>when problem size is large</u>
  - When problem size is small, “constant factors” matter
- A few common Orders of Growth:

| | | <u>Example</u>: |
|---|---|---|
| – Constant | O(1) | integer + integer |
| – Linear | O(n) | iterating through a list |
| – Quadratic | $O(n^2)$ | iterating through a grid |

# Example 1

```
def set_i_efficient(lst, i):
    lst[i] = 160

def set_i_inefficient(lst, i):
    for j in range(len(lst)):
        if j == i:
            lst[j] = 160
```

# Example 2

```
def make_pairs(list1, list2):
    """Return a list of pairs. Each pair is made
    of corresponding elements of list1 and list2.
    list1 and list2 must be of the same length."""
```

**So:** `make_pairs([2, 3, 4], ["x", "y", "z"])`

**Should return:** `[[2, "x"], [3, "y"], [4, "z"]]`

# Example 2

```
def make_pairs_linear(list1, list2):
    """Return a list of pairs. Each pair is made
    of corresponding elements of list1 and list2.
    list1 and list2 must be of the same length."""

    result = []
    for i in range(len(list1)):
        elt1 = list1[i]
        elt2 = list2[i]
        result.append([elt1, elt2])
    return result
```

# Example 2

```
def make_pairs_quadratic(list1, list2):
    """Return a list of pairs. Each pair is made
    of corresponding elements of list1 and list2.
    list1 and list2 must be of the same length."""

    result = []
    for i1 in range(len(list1)):
        elt1 = list1[i1]
        for i2 in range(len(list2)):
            elt2 = list2[i2]
            if i1 == i2:
                result.append([elt1, elt2])
    return result
```

# Running Times of Python Operations

Constant Time operations: O(1)

- Basic Math on numbers (+ - * /)
- Indexing into a sequence (eg. list, string, tuple) or dictionary
  - E.g. `myList[3] = 25`
- List operations: **`append`**, **`pop`**(at end of list)
- Sequence operation: **`len`**
- Dictionary operation: **`in`**
- Set operations: **`in`**, **`add`**, **`remove`**, **`len`**

Linear Time operations: O(n)

- **`for`** loop traversing an entire sequence or dictionary
- Built in functions: **`sum`**, **`min`**, **`max`**, slicing a sequence
- Sequence operations: **`in`**, **`index`**, **`count`**
- Dictionary operations: **`keys()`**, **`values()`**, **`items()`**
- Set operations: **`&`**, **`|`**, **`-`**
- String concatenation (linear in length of strings)

**Note**: These are general guidelines, may vary, or may have a more costly worst case. Built in functions (e.g. sum, max, min, sort) are often faster than implementing them yourself.