Announcements

• Ants project will actually be out in ~2 weeks
• Today:
  – One set of loose ends about mutability and lists
  – Understanding the Efficiency of code
Passing Data Into Functions
Learning Objectives

• Passing in a mutable object in a function in Python lets you modify that object
• Immutable objects don’t change when passed in as an argument
• Making a new name doesn’t affect the value outside the function
• Modifying mutable data **does** modify the values in the parent frame.
Mutating Input Data

• Functions can mutate objects passed in as an argument

• Declaring a new variable with the same name as an argument only exists within the scope of our function
  – You can think of this as creating a new name, in the same way as redefining a variable.
  – This will not modify the data outside the function, even for mutable objects.

• **BUT**
  – We can still directly modify the object passed in...even though it was created in some other frame or environment.
  – We directly call methods on that object.

• [View Python Tutor](http://cs88.org)
Python Gotcha’s: $a += b$ and $a = a + b$

- Sometimes similar *looking* operations have very different results!
- Why?
- $=$ always binds (or rebinds) a value to a name.
- $+=$ maps to the special method, e.g. `__iadd__`

```python
def add_data_to_obj(obj, data):
    print(obj)
    obj += data
    print(obj)
    return obj
def new_obj_with_data(obj, data):
    print(obj)
    obj = obj + data
    print(obj)
    return obj
```
Computational Structures in Data Science

Efficiency
Learning Objectives

• Runtime Analysis:
  – How long will my program take to run?
  – Why can’t we just use a clock?
  – How can we simplify understanding computation in an algorithm

• Enjoy this stuff? Take 61B!

• Find it challenging? Don’t worry! It’s a different way of thinking.
**Efficiency is all about trade-offs**

- Running Code: Takes Time, Requires Memory
  - More efficient code takes less time or uses less memory
- Any computation we do, requires both time and “space” on our computer.
- Writing efficient code is not obvious
  - Sometimes it is even convoluted!
- But!
- We need a framework before we can optimize code
- Today, we’re going to focus on the time component.
Is this code fast?

• Most code doesn’t *really* need to be fast! Computers, even your phones are already amazingly fast!

• Sometimes...it does matter!
  – Lots of data
  – Small hardware
  – Complex processes

• Slow code takes up battery power
Runtime analysis problem & solution

• Time w/stopwatch, but...
  – Different computers may have different runtimes. 😞
  – Same computer may have different runtime on the same input. 😞
  – Need to implement the algorithm first to run it. 😞

• Solution: Count the number of “steps” involved, not time!
  – Each operation = 1 step
    » 1 + 2 is one step
    » lst[5] is one step
  – When we say “runtime”, we’ll mean # of steps, not time!
Runtime: input size & efficiency

• Definition:
  - **Input size**: the # of things in the input.
  - e.g. length of a list, the number of iterations in a loop.
  - Running time as a function of input size
  - Measures **efficiency**

• Important!
  - In CS88 we won’t care about the efficiency of your solutions!
  - ... in CS61B we will
Runtime analysis: worst or average case?

• Could use avg case
  – Average running time over a vast # of inputs

• Instead: use worst case
  – Consider running time as input grows

• Why?
  – Nice to know most time we’d ever spend
  – Worst case happens often
  – Avg is often ~ worst

• Often called “Big O” for “order”
  – O(1), O(n) ...
Runtime analysis: Final abstraction

- Instead of an exact number of operations we’ll use abstraction
  – Want order of growth, or dominant term

- In CS88 we’ll consider
  – Constant
  – Logarithmic
  – Linear
  – Quadratic
  – Exponential

- E.g. $10n^2 + 4\log(n) + n$
  – ...is quadratic
Example: Finding a student (by ID)

• **Input**
  - Unsorted list of students L
  - Find student S

• **Output**
  - True if S is in L, else False

• **Pseudocode Algorithm**
  - Go through one by one, checking for match.
  - If match, true
  - If exhausted L and didn’t find S, false

• **Worst-case running time as function of the size of L?**
  1. Constant
  2. Logarithmic
  3. Linear
  4. Quadratic
  5. Exponential
Computational Patterns

• If the number of steps to solve a problem is always the same → Constant time: O(1)
• If the number of steps increases similarly for each larger input → Linear Time: O(n)
  – Most commonly: for each item
• If the number of steps increases by some a factor of the input → Quadratic Time: O(n²)
  – Most commonly: Nested for Loops
• Two harder cases:
  – Logarithmic Time: O(log n)
    » We can double our input with only one more level of work
    » Dividing data in “half” (or thirds, etc)
  – Exponential Time: O(2ⁿ)
    » For each bigger input we have 2x the amount of work!
    » Certain forms of Tree Recursion
Example: Finding a student (by ID)

• Input
  - Sorted list of students L
  - Find student S
• Output: same
• Pseudocode Algorithm
  - Start in middle
  - If match, report true
  - If exhausted, throw away half of L and check again in the middle of remaining part of L
  - If nobody left, report false

• Worst-case running time as function of the size of L?
  1. Constant
  2. Logarithmic
  3. Linear
  4. Quadratic
  5. Exponential
Comparing Fibonacci

```python
def iter_fib(n):
    x, y = 0, 1
    for _ in range(n):
        x, y = y, x+y
    return x

def fib(n): # Recursive
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

UC Berkeley | Computer Science 88 | Michael Ball | http://cs88.org
Tree Recursion

- \( \text{Fib}(4) \rightarrow 9 \text{ Calls} \)
- \( \text{Fib}(5) \rightarrow 16 \text{ Calls} \)
- \( \text{Fib}(6) \rightarrow 26 \text{ Calls} \)
- \( \text{Fib}(7) \rightarrow 43 \text{ Calls} \)
- \( \text{Fib}(20) \rightarrow \)
Why?

• Notice there was all this duplication in the tree?
• What is the exact order of growth?
  – It’s exponential.
  – phi to the N, where phi is the golden ratio.

<table>
<thead>
<tr>
<th>N</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>41</td>
</tr>
<tr>
<td>8</td>
<td>67</td>
</tr>
<tr>
<td>20</td>
<td>21891</td>
</tr>
</tbody>
</table>
Computational Structures in Data Science

Improving Efficiency
Learning Objectives

• Learn how to cache the results to save time.
• “memoization” is a specific version to avoid repeated calculations
Example

- Use a dictionary to cache results.
- This is called *memoization*

```python
def memo_fib(n):
    global fib_results
    if n in fib_results:
        print(f'found {n} -> {fib_results[n]}')
        return fib_results[n]
    if n < 2:
        fib_results[n] = n
        return n
    result = memo_fib(n - 1) + memo_fib(n - 2)
    fib_results[n] = result
    return result
```
A Better Approach

• Python’s `functools` module has a `cache` function
• [https://docs.python.org/3/library/functools.html#module-functools](https://docs.python.org/3/library/functools.html#module-functools)
• Uses a technique called decorators that we don’t cover.

```python
from functools import cache

def cache_fib(n):  # Recursive
    if n < 2:
        return n
    return cache_fib(n - 1) + cache_fib(n - 2)
```
What next?

• Understanding *algorithmic complexity* helps us know whether something is possible to solve.

• Gives us a formal reason for understanding why a program might be slow.

• This is only the beginning:
  – We’ve only talked about time complexity, but there is *space complexity*.
  – In other words: How much memory does my program require?
  – Often you can trade time for space and vice-versa.
  – Tools like “caching” and “memorization” do this.

• If you think this is cool take CS61B!