

# USABLE PROGRAMMING TOOLS FOR EXPERIMENTAL BIOLOGISTS

**Justin Lubin**

Advisor: Sarah E. Chasins  
EPIC Retreat, Spring 2022

**Vaccines**



**Precision  
health**



**Genomic  
editing**



**Wet lab**



**Dry lab**

**HANDS-ON  
PROTOCOLS**

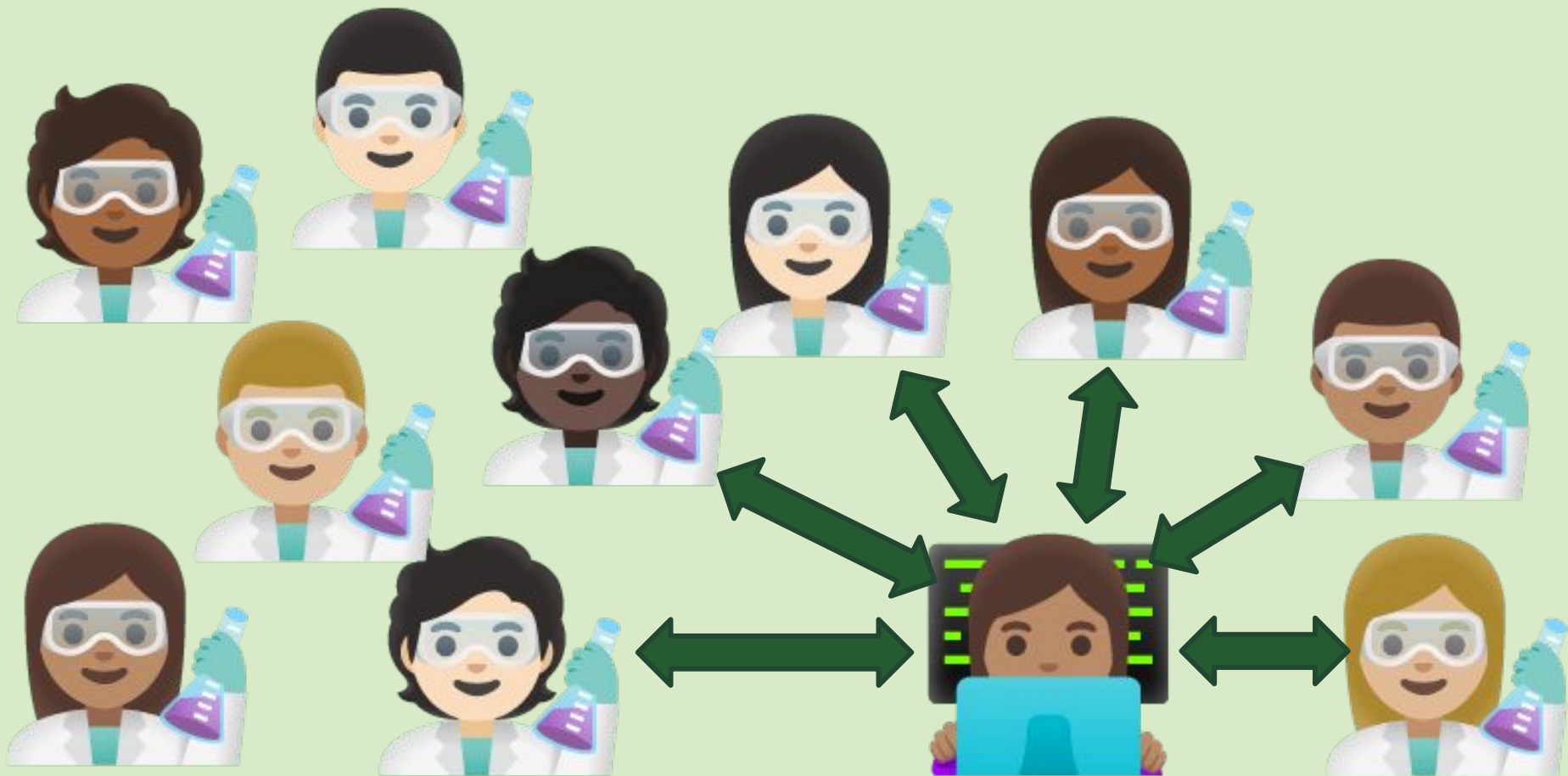
**Wet lab**



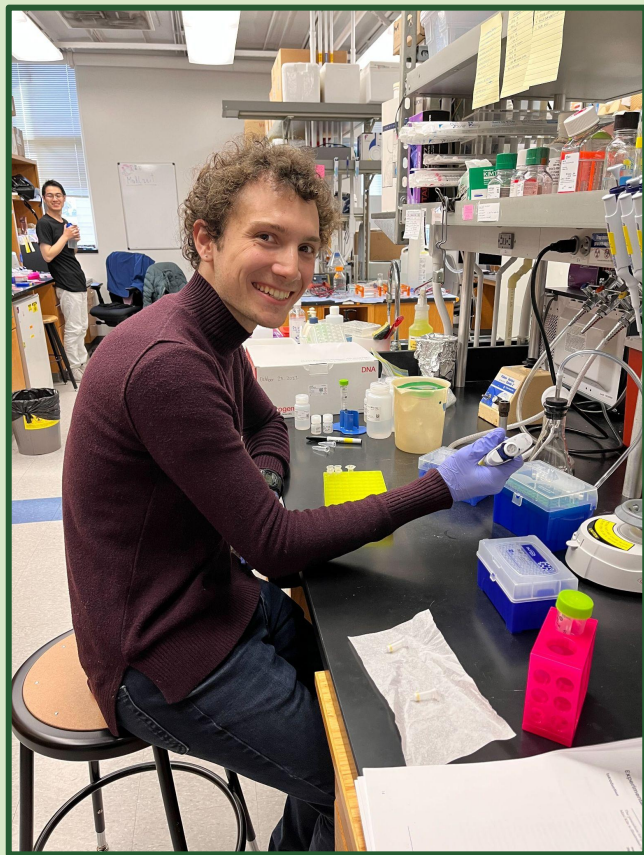
**CODE,  
MATH**

**Dry lab**











# USABLE PROGRAMMING TOOLS FOR EXPERIMENTAL BIOLOGISTS

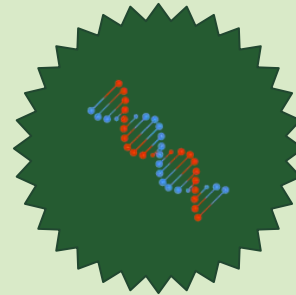
Vaccines



Precision health



Genomic editing



# COMPONENT-BASED REFACTORING

**WET LAB  
LANGUAGE**



# COMPONENT-BASED REFACTORING

WET LAB  
LANGUAGE

# COMPONENT-BASED REFACTORING

1. Motivating example
2. Naïve approach
3. Our approach

# COMPONENT-BASED REFACTORING

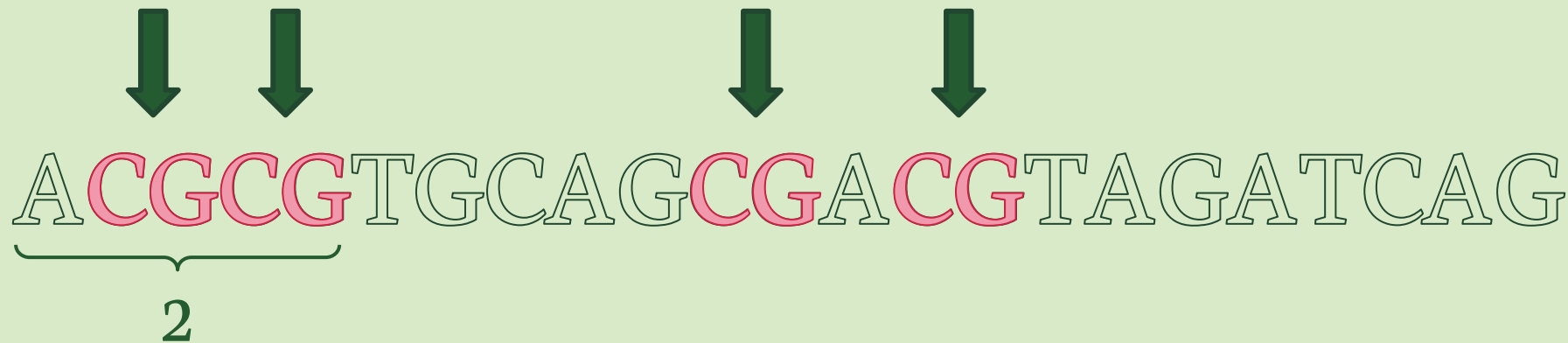
1. Motivating example
2. Naïve approach
3. Our approach

**ACGCGTGCAGCGACGTAGATCAG**



A diagram illustrating restriction enzyme recognition sites on a DNA sequence. The sequence is: ACGCGTGCA GCGACGTA GATCAG. The sequence is divided into three segments by spaces: ACGCGT, GCA G, and CGACGTA, followed by GATCAG. Four dark green arrows point downwards to specific recognition sites: the first two arrows point to the first 'CG' and the second 'CG' in the first segment (ACGCGT); the third arrow points to the 'CG' in the second segment (CGACGTA); and the fourth arrow points to the 'CG' in the third segment (CGTA). The 'CG' dinucleotides targeted by the arrows are highlighted in pink, while the other nucleotides are in light blue.

ACGCGTGCA GCGACGTA GATCAG



A diagram illustrating a DNA sequence: ACGCGTGCA GCGACG TAGATCAG. The sequence is written in a light green, outlined font. The first four bases (ACGC) are grouped by a bracket underneath, with the number 2 centered below the bracket. Four dark green arrows point downwards to specific bases: the first arrow points to the second base (C), the second arrow points to the third base (G), the third arrow points to the eighth base (C), and the fourth arrow points to the ninth base (A).

ACGCGTGCA GCGACG TAGATCAG

2

↓ ↓ ↓ ↓

A C G C G T G C A G C G A C G T A G A T C A G

2 2

The diagram shows a DNA sequence: A C G C G T G C A G C G A C G T A G A T C A G. The first four bases (A C G C) are grouped by a bracket with the number 2 below it. The next two bases (G T) are also grouped by a bracket with the number 2 below it. The remaining bases (G C A G C G A C G T A G A T C A G) are not grouped. Four dark green arrows point downwards to the bases: the first arrow points to the first 'C', the second to the second 'C', the third to the 'C' in the 'G C A' triplet, and the fourth to the 'C' in the 'A C G' triplet.

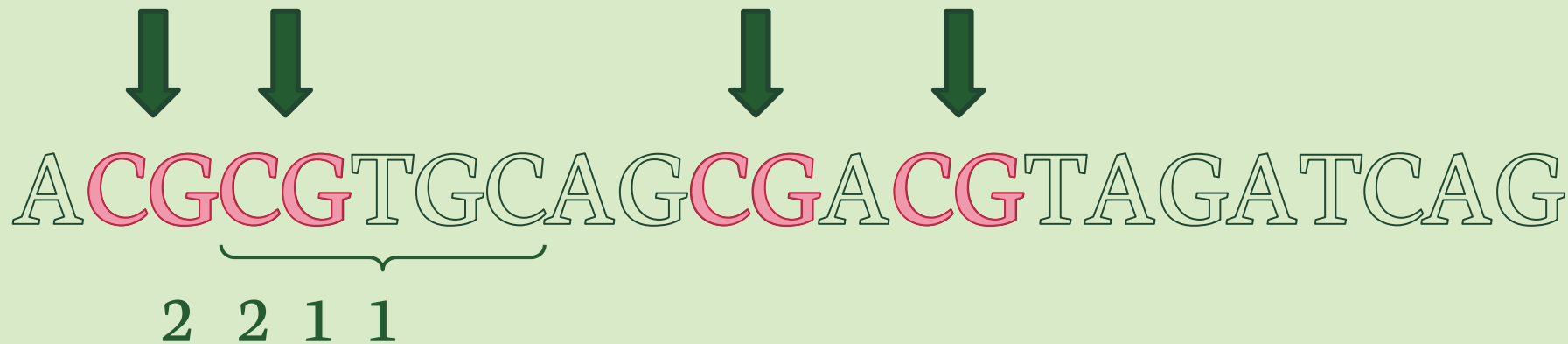


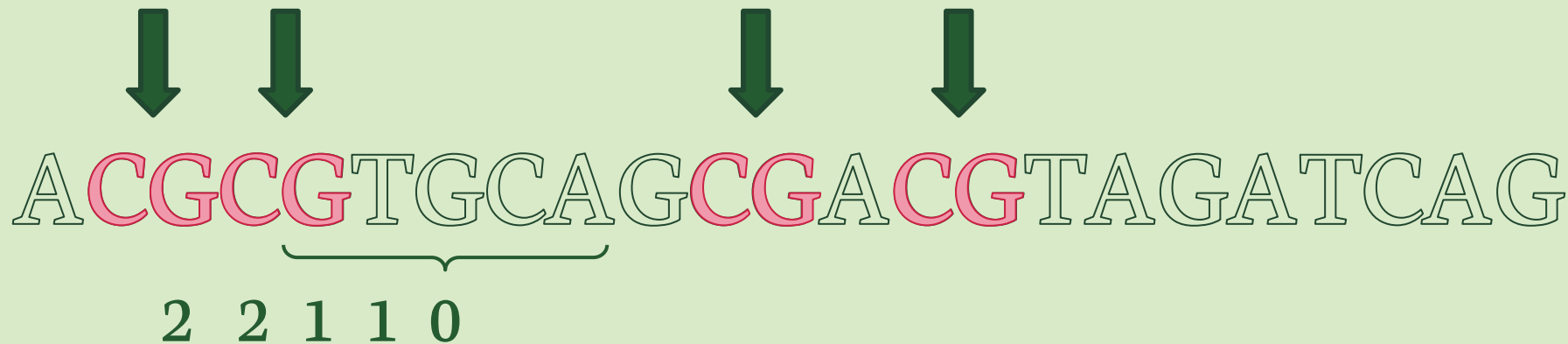
↓ ↓ ↓ ↓

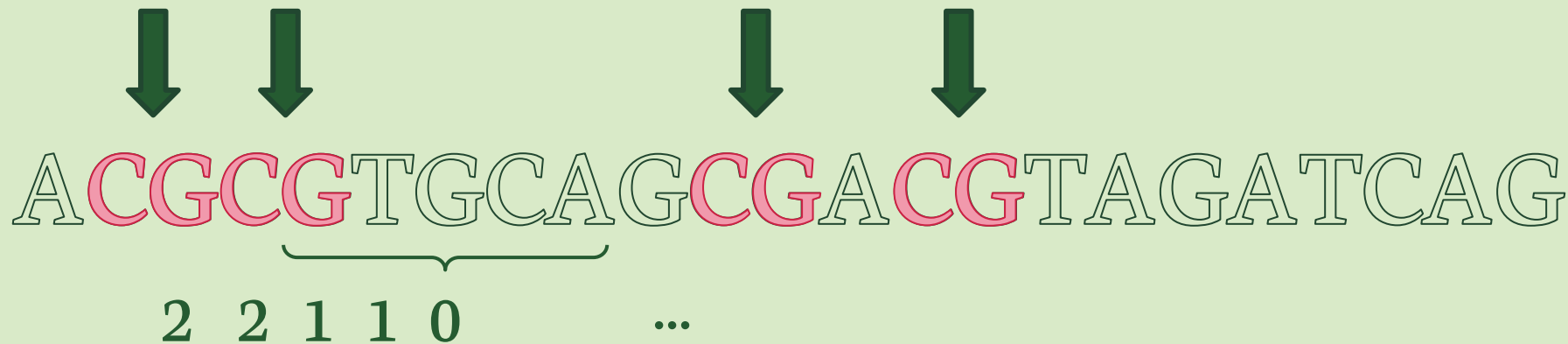
A C G C G T G C A G C G A C G T A G A T C A G

2 2 1

The diagram shows a DNA sequence: A C G C G T G C A G C G A C G T A G A T C A G. The first 'A' and the last 'G' are grey. The remaining 18 nucleotides are pink. Above the sequence, four dark green arrows point down to the 2nd, 3rd, 10th, and 13th positions. Below the first five nucleotides (A C G C G), a dark green bracket spans them, with the numbers 2, 2, and 1 positioned under the C, G, and C respectively.







↓ ↓ ↓ ↓

A C G C G T G C A G C G A C G T A G A T C A G

2 2 1 1 0 0 0 1 1 1 2 1 1 1 0 0 0 0 0

## Straightforward solution:

Two nested for loops



...but takes >1 day to run...

## Faster solution:

```
np.concatenate([
    np.convolve(
        (seq == "C")[:-1] & (seq == "G")[1:],
        np.ones(window_size),
        "valid"),
    [0]])
```

## Straightforward solution:

Two nested for loops



...but takes >1 day to run...

## Faster solution:

```
np.concatenate([  
    np.convolve(  
        (seq == "C")[:-1] & (seq == "G")[1:],  
        np.ones(window_size),  
        "valid"),  
    [0]])
```



...but takes <15 min to run!



# COMPONENT-BASED REFACTORING

1. Motivating example
2. Naïve approach
3. Our approach

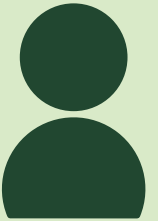
# COMPONENT-BASED REFACTORING

1. Motivating example
2. Naïve approach
3. Our approach



**sum(mul(x, y))**

```
def dot(x, y):  
    total = 0  
    for i in range(len(x)):  
        total += x[i] * y[i]  
    return total
```





`sum(mul(x, y))`

***Inline***



```
total = 0
for i in range(len(mul(x, y))):
    total += mul(x, y)[i]
return total
```

```
def dot(x, y):
    total = 0
    for i in range(len(x)):
        total += x[i] * y[i]
    return total
```





`sum(mul(x, y))`

*Inline*



```
total = 0
for i in range(len(mul(x, y))):
    total += mul(x, y)[i]
return total
```

```
def dot(x, y):
    total = 0
    for i in range(len(x)):
        total += x[i] * y[i]
    return total
```





`sum(mul(x, y))`

*Inline*



```
total = 0
for i in range(len(mul(x, y))):
    total += mul(x, y)[i]
return total
```

```
def dot(x, y):
    total = 0
    for i in range(len(x)):
        total += x[i] * y[i]
    return total
```



# COMPONENT-BASED REFACTORING

1. Motivating example
2. Naïve approach
3. Our approach



# COMPONENT-BASED REFACTORING

1. Motivating example
2. Naïve approach
3. **Our approach**



`sum(mul(x, y))`

*Inline*



```
total = 0
for i in range(len(mul(x, y))):
    total += mul(x, y)[i]
return total
```

```
def dot(x, y):
    total = 0
    for i in range(len(x)):
        total += x[i] * y[i]
    return total
```



`len(x)`

`x[i] * y[i]`

`len(mul(x, y))`

`mul(x, y)[i]`

# CANONICALIZATION

`len(mul(x, y))`  $\longrightarrow$  `len(x)`

`mul(x, y)[i]`  $\longrightarrow$  `x[i] * y[i]`



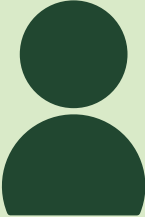
`sum(mul(x, y))`

*Canonicalize*



```
total = 0
for i in range(len(x)):
    total += x[i] * y[i]
return total
```

```
def dot(x, y):
    total = 0
    for i in range(len(x)):
        total += x[i] * y[i]
    return total
```





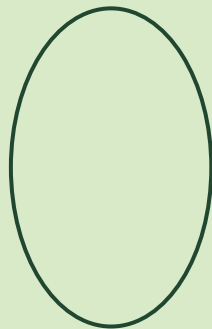
*Canonicalize*



*Canonicalize*

```
def dot(x, y):  
    total = 0  
    for i in range(len(x)):  
        total += x[i] * y[i]  
    return total
```

`sum(mul(x, y))`



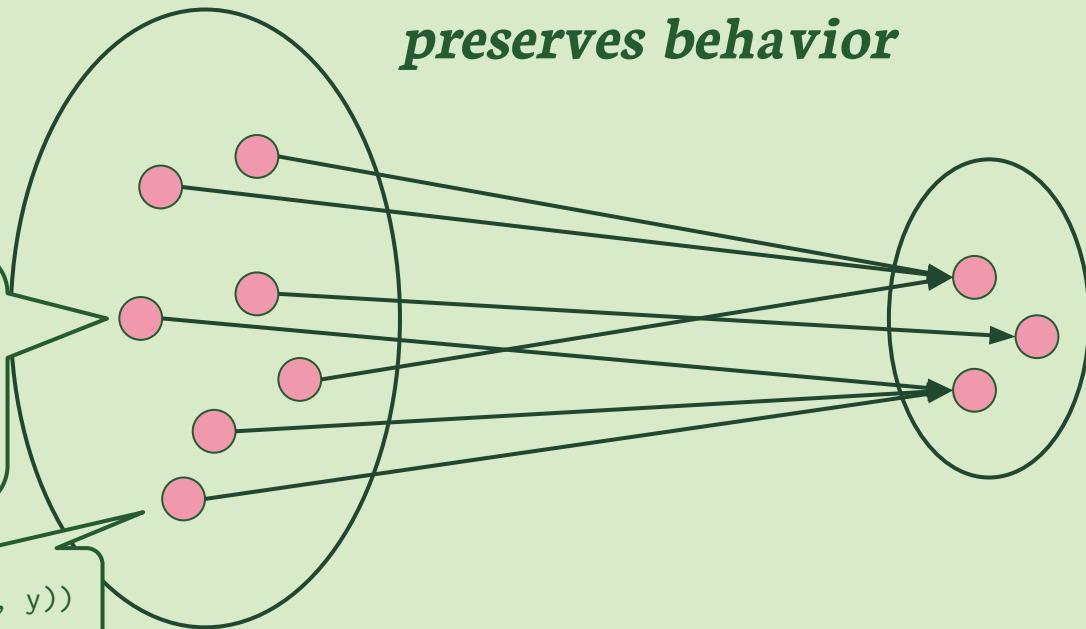


*Canonicalize*

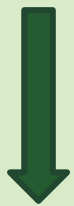
*preserves behavior*

```
def dot(x, y):  
    total = 0  
    for i in range(len(x)):  
        total += x[i] * y[i]  
    return total
```

```
sum(mul(x, y))
```

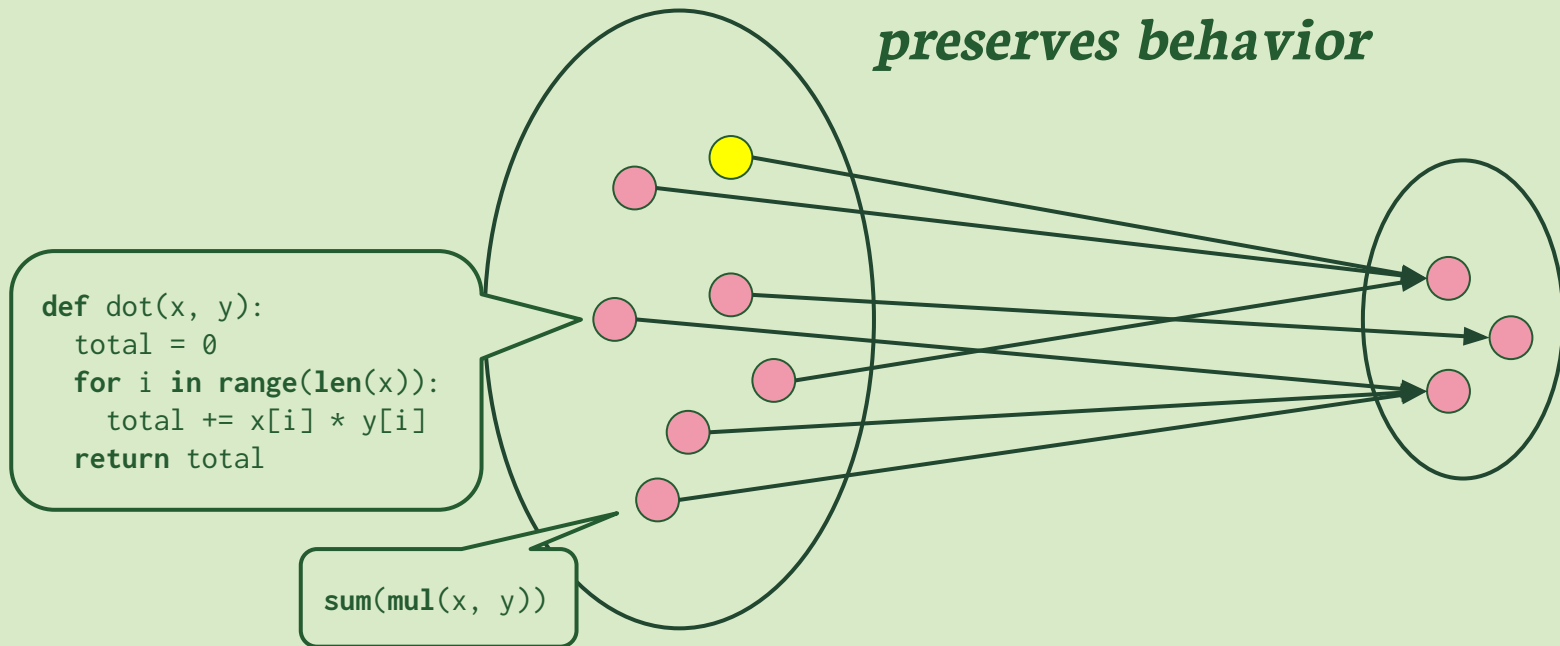






*Canonicalize*

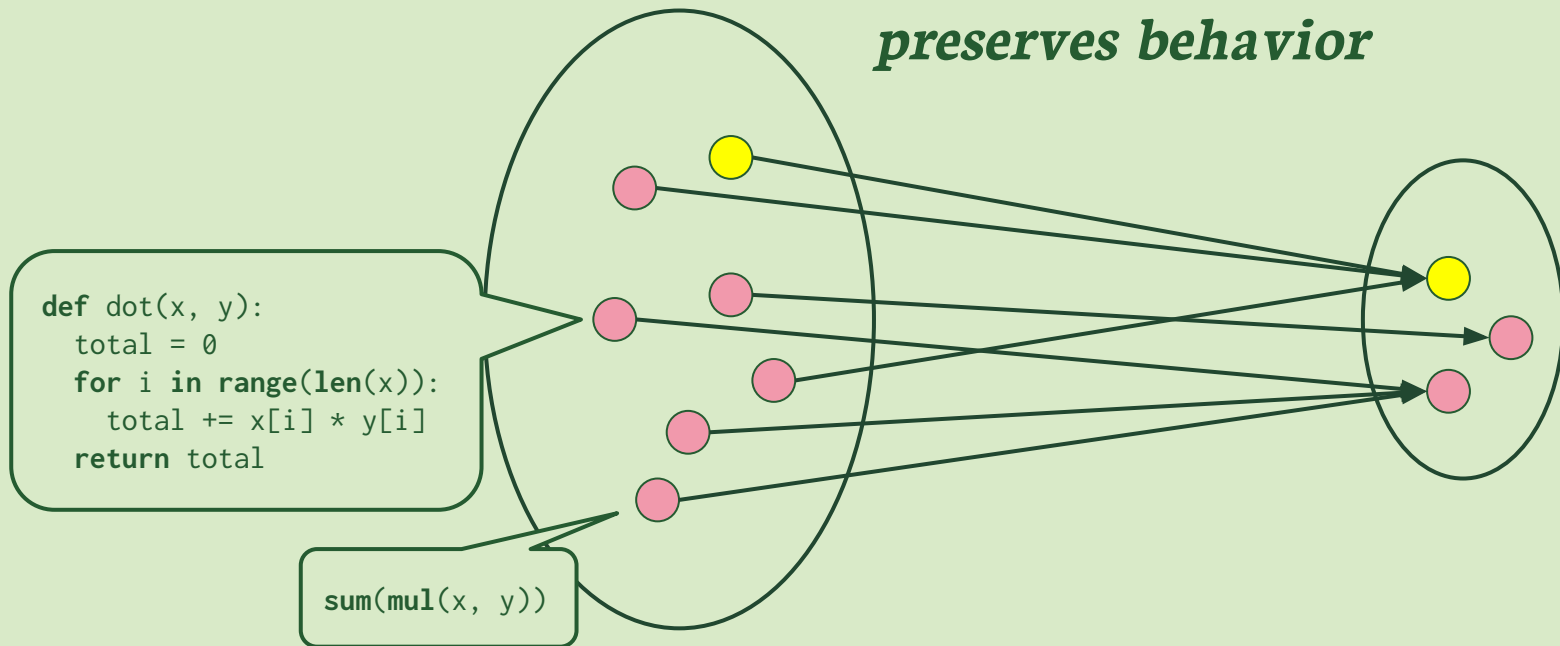
*preserves behavior*





*Canonicalize*

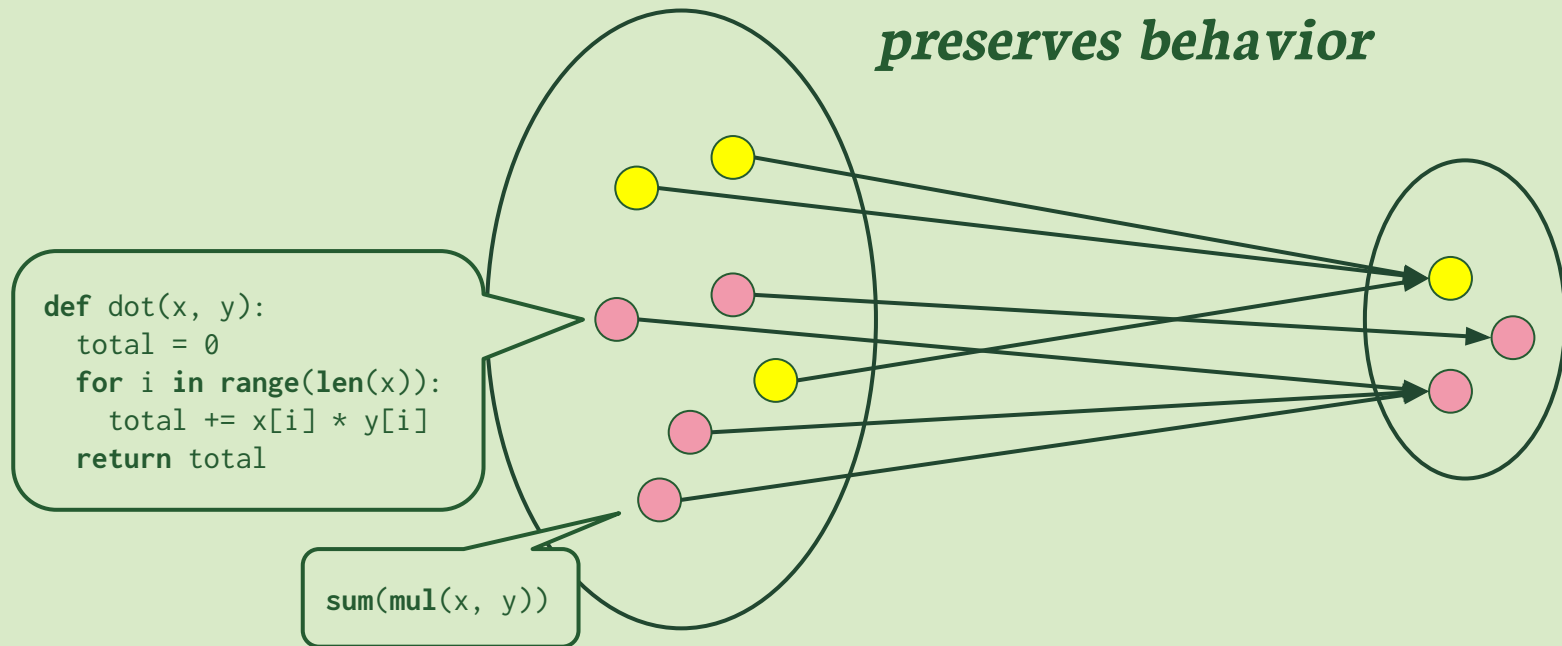
*preserves behavior*





*Canonicalize*

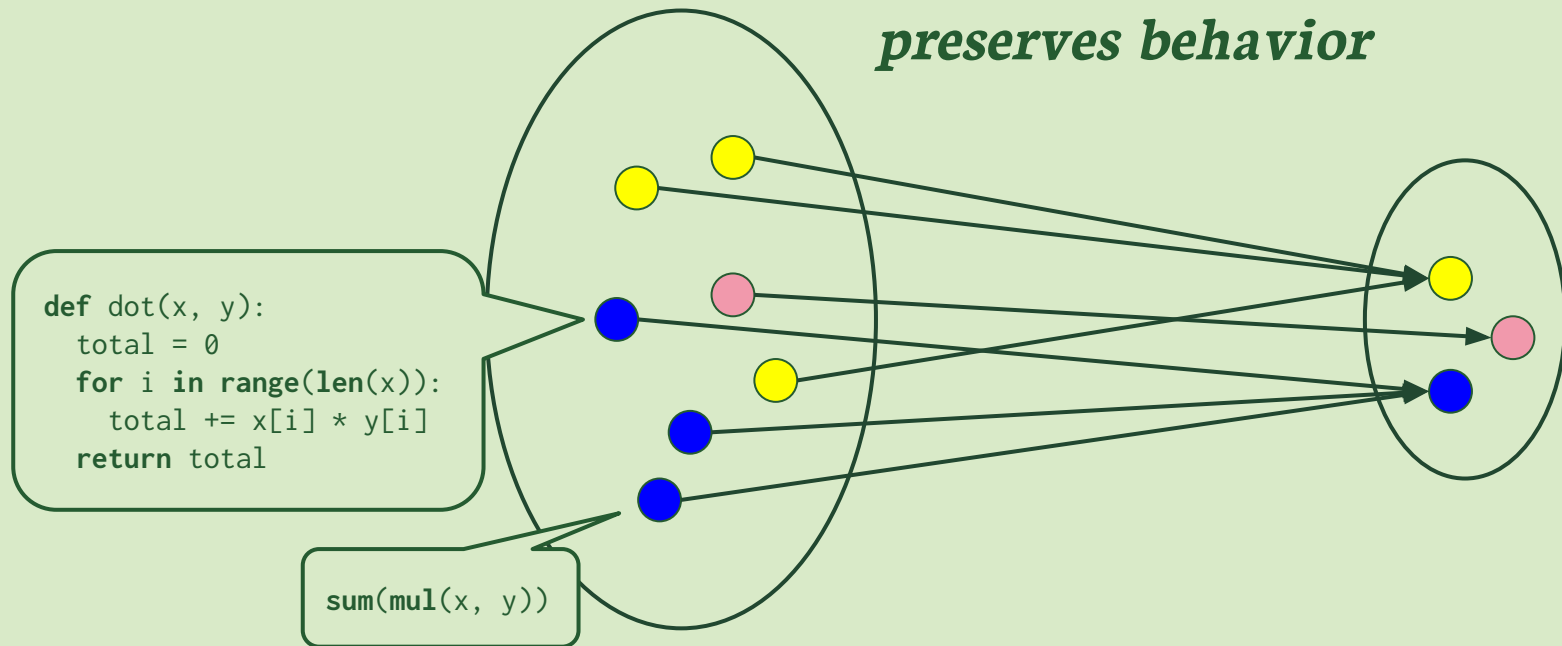
*preserves behavior*





*Canonicalize*

*preserves behavior*



For many programs,

**KEY**

**IDEA**

# KEY IDEA

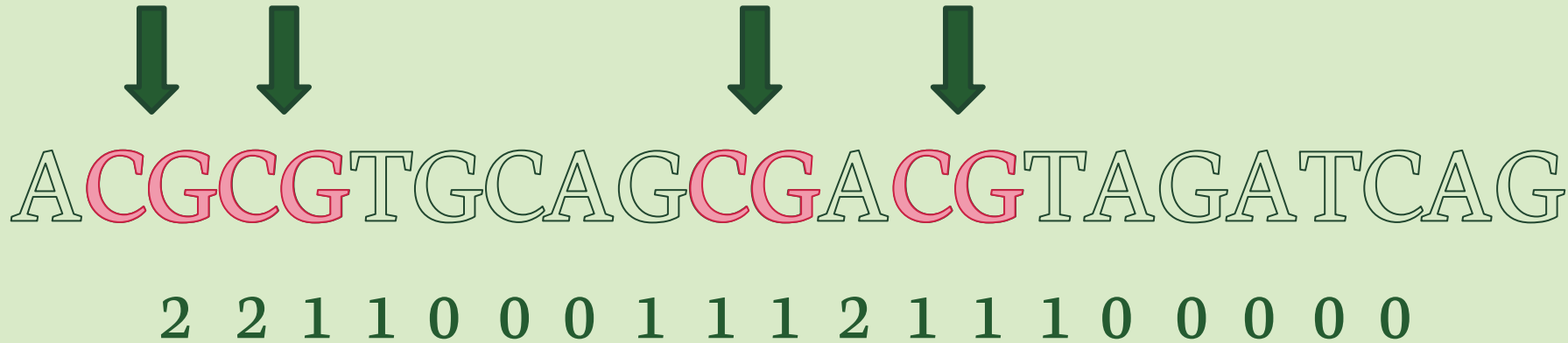
For many programs,  
we can check  
*extensional equality*

# KEY IDEA

For many programs,  
we can check  
*extensional equality*

— *by* —

*syntactic equality*  
modulo  
*canonicalization*



**Straightforward solution:**

Two nested for loops

*...takes **>1 day** to run...*

**Faster solution:**

```
np.concatenate([
    np.convolve(
        (seq == "C")[:-1] & (seq == "G")[1:],
        np.ones(window_size),
        "valid"),
    [0]])
```

*...takes **<15 min** to run!*





**Jeremy Ferguson**



**Kevin Ye**



**Jacob Yim**

↓ ↓ ↓ ↓

A C G C G T G C A G C G A C G T A G A T C A G

2 2 1 1 0 0 0 1 1 1 2 1 1 1 0 0 0 0 0

sum(mul(x, y))

Canonicalize

```
total = 0
for i in range(len(x)):
    total += x[i] * y[i]
return total
```

```
def dot(x, y):
    total = 0
    for i in range(len(x)):
        total += x[i] * y[i]
    return total
```

*preserves behavior*

