How to Code Fast(er)

How to Code Slow

1. Guess the “shape” of the code
2. Test the code
3. If it doesn’t work, make some random tweak
4. Repeat steps 2-3 until it works

How to Code Fast

1. Think *carefully* about structure and logic
2. Write the code *deliberately* – no guessing allowed
3. Test the code
4. If it doesn't work, *systematically* debug
   - If you don't know how to do this yet, watch this lecture on debugging from MIT OCW 6.000 at home later
   - DON'T run the code in your head to try to figure out what's wrong

How to Code Faster

1. Think *less*
2. Rely on good habits and patterns learned through practice

- Complex programs have many parts
- Thinking about all the parts at the same time is difficult for humans
- Luckily, we invented ways to write programs
    1. Without needing to think about some parts at all
    2. Without needing to think about all the parts at the same time
    3. Without doing the steps that the computer is supposed to be the one doing

How to Code Fast(er)
Part 1: Use Patterns

Let’s Write Some Code!

1. Given a list of integers, print the square of each integer
2. Given a string of uppercase letters, print it in lower case

Notice Anything Similar?

```python
def square(x):
    return x * x
```

```python
def lower(c):
    return (chr(ord(c)
            - ord('A')
            + ord('a')))
```

```python
for x in numbers:
    print(square(x))
```

```python
for c in s:
    print(lower(c), end='')
```

The Map Pattern

```
for item in input_sequence:
    apply_operation(item)
```

- Replace red parts with relevant code

The Map Pattern

- Given:
  - Input sequence
  - Operation to perform
- Do:
  - Perform the operation on each item of the sequence
- Produce:
  - A new list containing the results of each operation

Define a Function That Captures This Pattern

```python
def map(operation, sequence):
    result = []
    for item in sequence:
        result.append(operation(item))
    return result
```

We Can Rewrite Our Previous Examples

```
def square(x):
    return x * x


print(map(square, numbers))
```

```
def lower(c):
    return (chr(ord(c)
                - ord('A')
                + ord('a')))

print(''.join(map(lower, s)))
```

Wait, But Why?

- You were probably taught how to think of loops in terms of all the individual steps
- That's too complicated: instead, focus on defining the operation for just **one element**
- In Python, the `map` function is already built-in
- In C++, type a for-loop but think of `map`

Carefully Think About and Focus on the Parts One-at-a-Time

- First, think about the *high-level*, *overall* goal
- Name and use the function you need to put into map, even if they are not defined yet

```
print(map(square, numbers))
print(''.join(map(lower, s)))
```

- Worry about their definition later

Carefully Think About and Focus on the Parts One-at-a-Time

```python
def square(x):
    return x * x
```

```python
def lower(c):
    return (chr(ord(c)
              – ord('A')
              + ord('a')))
```

- These are the really important parts of the problem; the for-loop is a “minor detail” you can worry about later, or not worry about at all
- The map pattern encourages you to *separately* think about
  - The operation you want to perform on each element
  - The mechanics of going through elements one-by-one

Boring? Ok, Here’s An Exercise

- Given a list of lists, create a new list containing the reverse of each

- Sample Input

```
[[1, 2, 3],
 [4, 5, 6, 7],
 [8, 9]]
```

- Sample Output

```
[[3, 2, 1],
 [7, 6, 5, 4],
 [9, 8]]
```

Solution

```
map(reversed, list_of_lists)
```

Solution

```
def reversed(a):
    ans = []
    index = len(a) - 1
    while index >= 0:
        ans.append(a[index])
        index -= 1
    return ans
```

- Fancier ways exist in Python, but we won’t show them here

No Need to Literally Write a Separate Function

```
result = []
for item in input_sequence:
    # apply the operation to item here
    # save the result to “transformed_item”
    result.append(transformed_item)
```

- When the operation is simple enough, the map function can be “in your head” only and not “in your code”

No Need to Literally Write a Separate Function: This Is Closer to What You Might Write in C++

```
result = []
for a in list_of_lists:
    reversed = []
    index = len(a) - 1
    while index >= 0:
        reversed.append(a[index])
        index -= 1
    result.append(reversed)
```

Let’s Write More Code!

1. Given a list of integers, print the ones that are even
2. Given a list of integers, print the ones that are between two given integers a and b

Notice Anything Similar?

```python
def is_even(x):
    return x % 2 == 0

for x in nums:
    if is_even(x):
        print(x)
```

```python
def in_range(x):
    return a <= x <= b

for x in nums:
    if in_range(x):
        print(x)
```

The Filter Pattern

```
for item in input_sequence:
    if condition_holds_for(item):
        apply_operation(item)
```

- Replace red parts with relevant code

The Filter Pattern

- Given:
  - Input sequence
  - Condition to check
- Do:
  - Check if condition holds for each item of the sequence
- Produce:
  - A new, filtered list containing only those items which satisfy the condition

Define a Function That Captures This Pattern

```
def filter(condition, sequence):
    result = []
    for item in sequence:
        if condition(item):
            result.append(item)
    return result
```

We Can Rewrite Our Previous Examples

```
def is_even(x):
    return x % 2 == 0

print(filter(is_even, nums)))
```

```
def in_range(x):
    return a <= x <= b

print(filter(in_range, nums))
```

Again…

- You were probably taught how to think of loops in terms of all the individual steps
- That's too complicated: instead, focus on defining a test for just **one element**
- In Python, the `filter` function is already built-in
- In C++, type a for-loop + if but think of `filter`

Exercise

- Given a string, print only the letters – no spaces, digits, punctuation

- Sample Input

```
Hello, World! 123
```

- Sample Output

```
HelloWorld
```

Solution

```
filter(is_alpha, input_string)
```

Solution

```
def is_alpha(c):
    return 'A' <= c <= 'Z' or 'a' <= c <= 'z'
```

Without the Filter Function

```
for c in input_string:
    if 'A' <= c <= 'Z' or 'a' <= c <= 'z':
        print(c, end='')
```

Let’s Write More Code!

1. Given a list of integers, print their sum
2. Given a list of integers, print their product
3. Given a list of integers, print their maximum
   - Assume all integers are < 1000 in absolute value
4. Given a list of integers, print their minimum
   - Assume all integers are < 1000 in absolute value

Notice Anything Similar?

```
ans = 0
for x in input_list:
    ans = ans + x
```

```
ans = -1000
for x in input_list:
    ans = max(ans, x)
```

```
ans = 1
for x in input_list:
    ans = ans * x
```

```
ans = 1000
for x in input_list:
    ans = min(ans, x)
```

The Reduce Pattern

```
ans = identity
for item in input_sequence:
    ans = binary_operation(ans, item)
```

- Replace red parts with relevant code

The Reduce Pattern

- Given:
  - Input sequence
  - Binary operation to perform
  - The *identity* element of the binary operation
- Do:
  - Successively apply the binary operation to successive elements of the list
    - Taking as left operand the result of the previous application (initially just the identity) and as right operand the current element of the list
- Produce:
  - The result of doing the successive application

Identity Element?

- The identity of a binary operation $\otimes$ is some value e such that

$$e \otimes x = x \otimes e = x$$

for all x in the domain of the operation

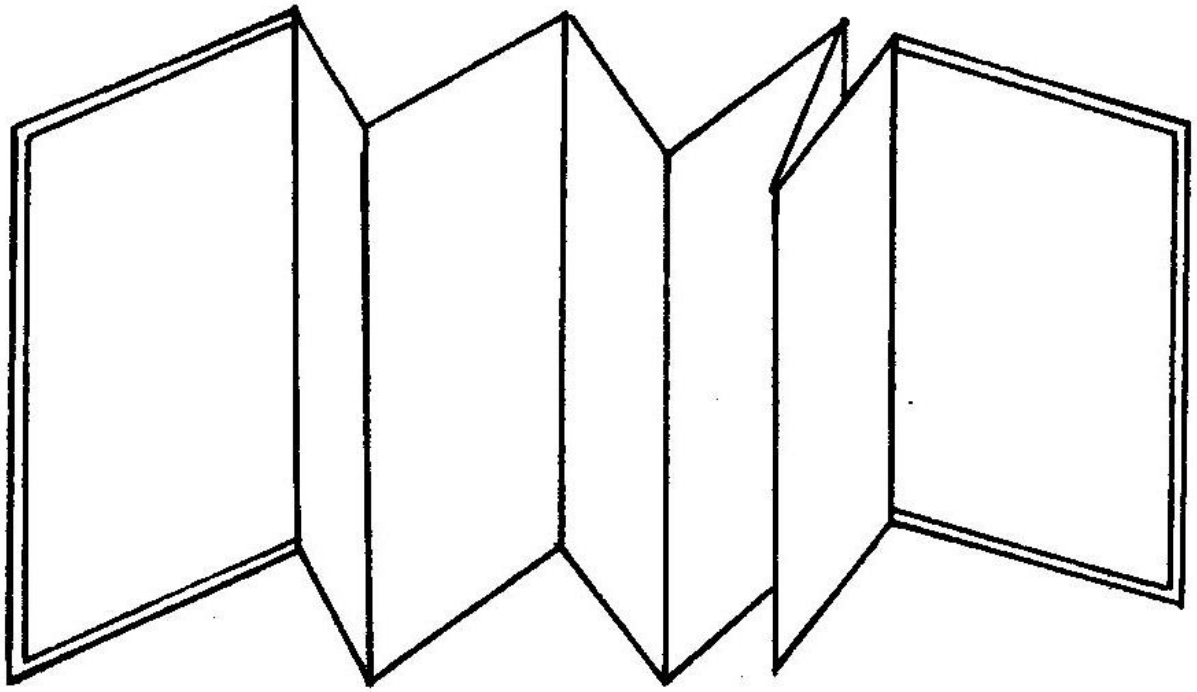
- Examples:
  - The *additive identity* for $+$ is 0 because $0 + x = x + 0 = x$
  - The *multiplicative identity* for $*$ is 1 because $1 * x = x * 1 = x$
  - The identity for $\min$ is ∞ because $\min(\infty, x) = \min(x, \infty) = x$
    - In practice, we just define ∞ as some very large number, larger than any input we might encounter, because ∞ does not really exist

Another Way to Think About It

- The reduce operation *generalizes* a binary operation that works for only two elements into an n-ary operation that works for any number of elements

AKA **Fold**

- Imagine each part is an element of array
- Folding together two parts is applying binary operation



Define a Function That Captures This Pattern

```python
def reduce(binary_operation, sequence, identity):
    result = identity
    for item in sequence:
        result = binary_operation(result, item)
    return result
```

We Can Rewrite Our Previous Examples

```
def plus(x, y):
    return x + y

def times(x, y):
    return x * y

reduce(plus, input_list, 0)
reduce(times, input_list, 1)
reduce(max, input_list, -1000)
reduce(min, input_list, 1000)
```

Again…

- You were probably taught how to think of loops in terms of all the individual steps
- That’s too complicated: instead, focus on how to combine only **two elements**
- In Python, the `reduce` function is already built-in
    - In Python 3, need to do `from functools import reduce`
- In C++, type a for-loop but think of `reduce`

Exercise

- Given a list of lists, *flatten* it: combine all elements to a single list
- Assume the inner lists contain only numbers

- Sample Input

```
[[1, 2, 3],
 [4, 5, 6, 7]
 [8, 9]]
```

- Sample Output

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Solution

```
reduce(concatenate, list_of_lists, [])
```

Solution

```python
def concatenate(a, b):
    return a + b

def concatenate(a, b): # without Python magic
    ans = []
    for x in a:
        ans.append(x)
    for x in b:
        ans.append(x)
    return ans
```

Without the Reduce Function

```
result = []
for a in list_of_lists:
    result = result + a
```

Without Python Magic

```
result = []
for a in list_of_lists:
    temp = []
    for x in result:
        temp.append(x)
    for x in a:
        temp.append(x)
    result = temp
```

More Efficient Version

```
result = []
for a in list_of_lists:
    for x in a:
        result.append(x)
```

Exercise

- Write an alternate version of reduce that doesn’t require an identity, instead requires sequence to be non-empty

Solution

```python
def reduce(binary_operation, sequence):
    result = sequence[0]
    for item in sequence[1:]:
        result = binary_operation(result, item)
    return result

# solution without Python slicing magic is also ok
```

How to Use These Patterns

- Thinking about flow of control can be too complicated, especially with nested structures
- If operation you want to perform follows patterns, it’s easier to think about operating on the list as a whole
- Stop thinking about all the individual steps
  - Map: just think about operation to perform
  - Filter: just think about condition to test
  - Reduce: just think about binary operation and identity
- The for-loop becomes “just syntax” and you can focus on filling in the parts that really matter for your problem

How to Use These Patterns

- In a more advanced language like Python, you may want to literally use map, filter, reduce – it leads to cleaner, shorter code
- In an uglier language like C++, just write the for-loops, but in your mind, if you think about map, filter, reduce, you may find it easier to write the loop

Exercise

- Define a function called **`any`**, which take a sequence of Booleans and returns `True` if any one of them are `True`, `False` otherwise
  - Using a regular loop
  - Using reduce
- Define a function called **`all`**, which take a sequence of Booleans and returns `True` if all of them are `True`, `False` otherwise
  - Using a regular loop
  - Using reduce

Solution: Using Regular Loops

```
def any(sequence):
    ans = False
    for item in sequence:
        ans = ans or item

def all(sequence):
    ans = True
    for item in sequence:
        ans = ans and item
```

Solution: Using Reduce

```python
def OR(x, y):
    return x or y

def AND(x, y):
    return x and y

def any(sequence):
    return reduce(OR, sequence, False)

def all(sequence):
    return reduce(AND, sequence, True)
```

Lambda Expressions

- It can be quite awkward to have to write function definitions for extremely simple functions, like OR and AND above
- Sometimes, more natural to supply the entire definition of a function in a single expression, without naming it
- A **lambda expression** allows us to do just this, but only for really simply functions that have only the return expression in its body
  - Otherwise, the function is actually complicated and would be better defined and named elsewhere
- Syntax: `lambda <parameters>: <return expression>`

`Any` and `All`, with Lambda Expressions

```python
def any(sequence):
    return reduce(lambda x, y: x or y, sequence, False)

def all(sequence):
    return reduce(lambda x, y: x and y, sequence, True)
```

List Comprehensions

- Map/filter with lambda syntax can be ugly, so Python offers an alternative syntax

```
map(lambda x: <expression>, sequence)
```

can be written as

```
[<expression> for x in sequence]
```

Example: Create a List of Perfect Squares

```
[x * x for x in range(1, n)]
[square(x) for x in range(1, n)] # also valid
```

List Comprehensions

```
filter(lambda x: <condition>, sequence)
```

can be written as

```
[x for x in sequence if <condition>]
```

Example: Make a New List Containing Non-Negative Elements of Given List

```
[x for x in sequence if x > 0]
[x for x in sequence if is_positive(x)] # also valid
```

List Comprehensions

- Of course, they can be combined

```
[<expression> for x in sequence if <condition>]
```

is the same as

```
map(expression, filter(condition, sequence))
```

Example

```
[x * x for x in sequence if x > 0]
```

Note

- For reasons that will take us too long to explain, the expression

```
<expression> for x in sequence if <condition>
```

is called a **generator expression**

- If you see a `SyntaxError` mentioning this, this is what it refers to
- As long as you enclose it in brackets, it automatically becomes a list and you can deal with it like any other list (example, for printing out)

Exercises

1. Write a program which given a list of numbers and a number x, determines whether x is in the list
2. Write a program which given a list of numbers, determines whether all numbers are positive
3. Write a program to check if an integer ≥ 2 is prime
   - x is a **divisor** of y if the remainder after dividing y by x is 0
   - A number is **prime** if it has no divisors other than 1 and itself

Solutions

```
def search(sequence, x):
    def is_x(elem):
        return elem == x
    return any(map(is_x, sequence))
```

Or Simply

```python
def search(sequence, x):
    return any(map(lambda elem: elem == x, sequence))

def search(sequence, x):
    return any(elem == x for elem in sequence)
```

Solutions

```python
def all_positive(sequence):
    def is_positive(elem):
        return elem > 0
    return all(map(is_positive, sequence))

def all_positive(sequence):
    return all(map(lambda elem: elem > 0, sequence))

def all_positive(sequence):
    return all(elem > 0 for elem in sequence)
```

Solutions

```
def is_prime(n):
    def is_divisor(x):
        return n % x == 0
    return not any(map(is_divisor, range(2, n)))

def is_prime(n):
    return not any(n % x == 0 for x in range(2, n))
```

Alternatively

```
def is_prime(n):
    def not_divisor(x):
        return n % x != 0
    return all(map(not_divisor, range(2, n)))

def is_prime(n):
    return all(n % x != 0 for x in range(2, n)))
```

Important Note

- You might not need to literally use `map`, `any`, `all`
  - Especially in C++ where there are equivalents, but uglier
- But you should be able to recognize `any` and `all` as patterns and be able to very quickly write loops that essentially perform some variant of these

The Any Pattern

- Given
  - Input sequence
  - Condition to check
- Do:
  - Check if condition holds for each item of the sequence
- Produce:
  - `True` if there is at least one item of the sequence satisfying the condition, `False` otherwise

The Any Pattern

```
ans = False
for item in input_sequence:
    if condition_holds_for(item):
        ans = True
```

- Replace red parts with relevant code

Alternatively

```
ans = False
for item in input_sequence:
    ans = ans or condition_holds_for(item)
```

The All Pattern

- Given
  - Input sequence
  - Condition to check
- Do:
  - Check if condition holds for each item of the sequence
- Produce:
  - `False` if there is at least one item of the sequence violating the condition, `True` otherwise

The All Pattern

```
ans = True
for item in input_sequence:
    if not condition_holds_for(item):
        ans = False
```

- Replace red parts with relevant code

Alternatively

```
ans = True
for item in input_sequence:
    if condition_fails_for(item):
        ans = False
```

Alternatively

```
ans = True
for item in input_sequence:
    ans = ans and condition_holds_for(item)
```

Any and All Comparison

Any

```
ans = False
for item in sequence:
  if condition(item):
    ans = True
```

All

```
ans = True
for item in sequence:
  if not condition(item):
    ans = False
```

Questions

- Given an empty list, are any of its elements equal to zero?
- Given an empty list, are all of its elements equal to zero?

Answers

- Given an empty list, are any of its elements equal to zero?
  - No, given any condition and an empty list, it is *trivially false* that some element in the list that satisfies the condition
- Given an empty list, are all of its elements equal to zero?
  - Yes, given any condition and an empty list, it is [*vacuously true*] that all elements in the list satisfy the condition
  - Explanation: such questions never make sense in real life, but the correct answer is defined this way for mathematical convenience

Rewrite These Three Examples Using Regular Loops

1. Write a program which given a list of numbers and a number x, determines whether x is in the list
2. Write a program which given a list of numbers, determines whether all numbers are positive
3. Write a program to check if an integer ≥ 2 is prime
   - x is a **divisor** of y if the remainder after dividing y by x is 0
   - A number is **prime** if it has no divisors other than 1 and itself

Solutions

```
def search(sequence, x):
    ans = False
    for elem in sequence:
        if elem == x:
            ans = True
    return ans
```

Solutions

```python
def all_positive(sequence):
    ans = True
    for elem in sequence:
        if not elem > 0:
            ans = False
    return ans
```

Solutions

```python
def is_prime(n):
    composite = False
    for x in range(2, n):
        if n % x == 0:
            composite = True
    return not composite
```

Alternatively

```
def is_prime(n):
    ans = True
    for x in range(2, n):
        if n % x == 0:
            ans = False
    return ans
```

Patterns Can Be Combined to Do More Interesting Things

- Print the sum of the squares of the prime numbers from 2 to 100

Solution Sketch

`[2, 3, 4, 5, …, 99]`

Filter: number must be prime

`[2, 3, 5, …, 97]`

Map: operation is squaring

`[4, 9, 25, …, 9409]`

Reduce: operation is +

Answer

Solution

```
input_sequence = range(2, 100)
primes = filter(is_prime, input_sequence)
squared_primes = map(square, primes)
ans = reduce(plus, squared_primes)
```

Solution

```
input_sequence = range(2, 100)
primes = filter(is_prime, input_sequence)
squared_primes = map(square, primes)
ans = reduce(plus, squared_primes)
```

Or, more concisely…

```
reduce(plus, map(square, filter(is_prime, range(2, 100))))
```

Sum of the squares of the prime numbers between 2 and 100

How to Do It Without Python Magic

```
input_sequence = range(2, 100)
primes = []                                              Filter
for x in input_sequence:
    is_prime = True                                 All
    for i in range(2, x):
        is_prime = is_prime and x % i != 0
    if is_prime:
        primes.append(x)
```

How to Do It Without Python Magic

Map

```
squared_primes = []
for p in primes:
    squared_primes.append(p * p)
```

Reduce

```
ans = 0
for sp in squared_primes:
    ans = ans + sp
```

How to Do It Without Python Magic

- You don’t have to literally stick to the patterns
- The patterns are there to help you think
- The code doesn’t necessarily have to be a mechanical translation of the patterns, but the patterns will “still be there”

How to Do It Without Python Magic

```
ans = 0
for x in range(2, 100):
    is_prime = True
    for i in range(2, x):
        is_prime = is_prime and x % i != 0
    if is_prime:
        ans += x * x
```

How to Do It Without Python Magic

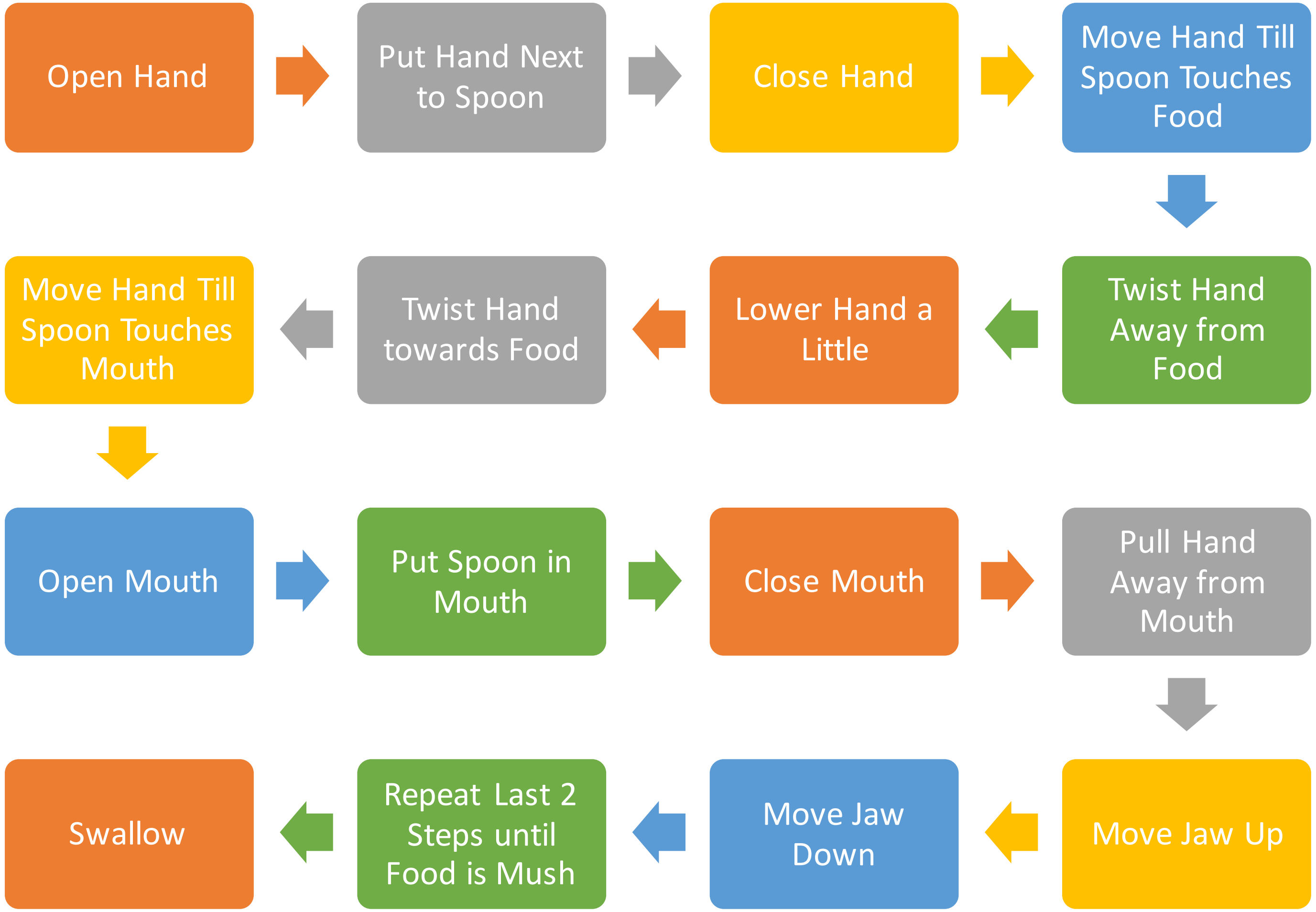
- This code seems scary and complicated…
- But what we really want to do is simple

```
reduce(plus, map(square, filter(is_prime, range(2, 100))))
```

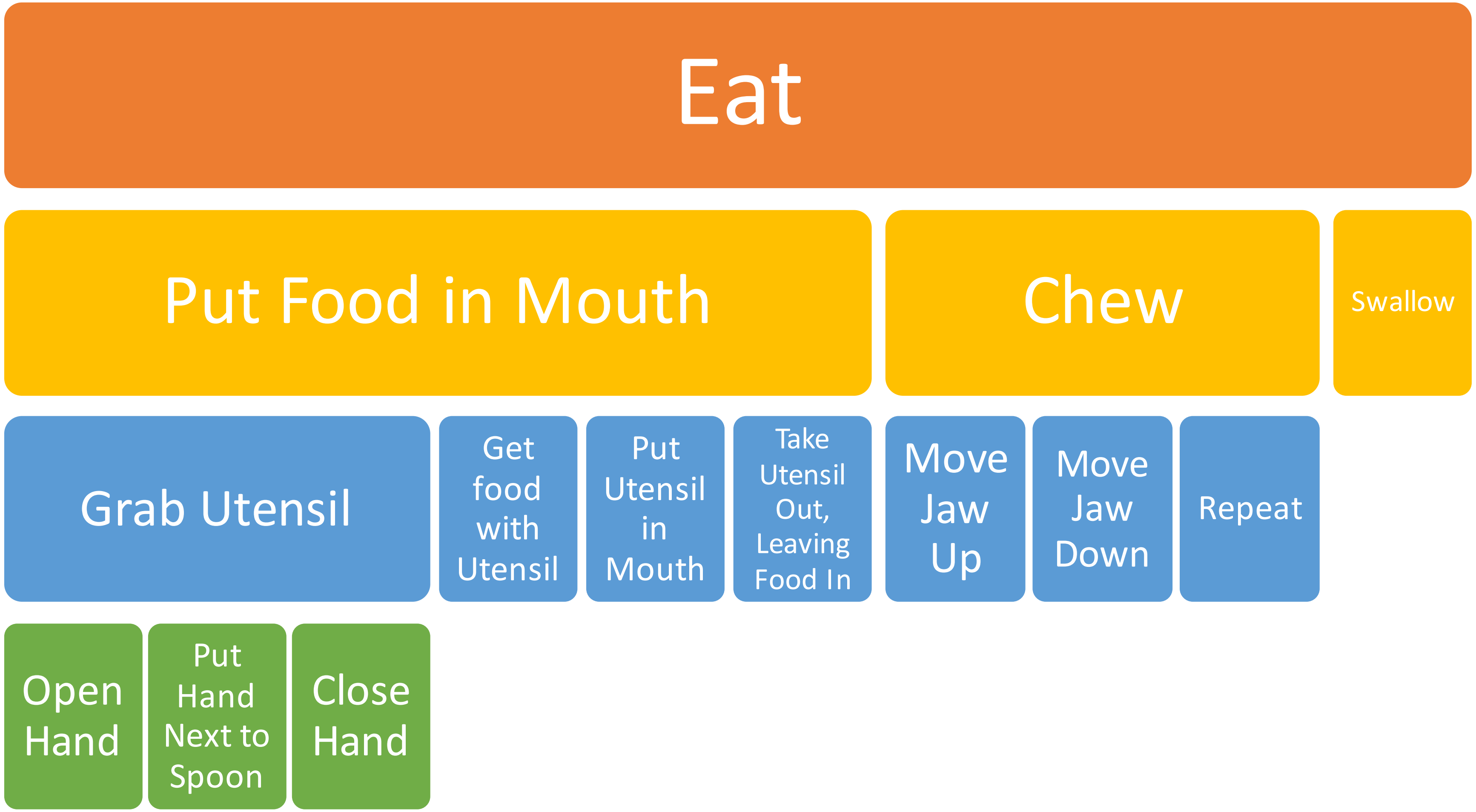
- Think about the idea first
  - How would you explain it to a *human*, not a computer?
- Then, think of the code as just a *translation* of the idea
- Then, it won't feel scary and complicated

How to Code Fast(er)
Part 2: Top-Down Design

Teaching a Computer to Eat



Teaching a Computer to Eat



Two Different Ways to Say the Same Thing…

```
reduce(plus, map(square, filter(is_prime, range(2, 100))))

ans = 0
for x in range(2, 100):
    is_prime = True
    for i in range(2, x):
        is_prime = is_prime and x % i != 0
    if is_prime:
        ans += x * x
```

Two Different Ways to Say the Same Thing…

High-level

- Closer to "human" way of expressing it
- Less detailed

Low-level

- Closer to "primitive" operations available on the computer
- More detailed

- Can be many levels in between

What Makes Programming Complex?

- Task to be done consists of many low-level steps
- It would be much easier if programming language already had high-level functions available
- Computers and prog. languages are designed to be **general-purpose**: people working on different problems can use the same set of tools
  - A single effort to improve the tools benefits *everyone*
  - Much cheaper to build computer with as few available operations as possible
  - Not everyone will need the high-level function that you need
- A general-purpose tool is necessarily low-level

Fortunately, There's a Way to Handle This Complexity

- DON'T directly solve your problem using the low-level tools
- *Invent* new high-level operations (language) suitable for your problem
- This is what defining functions/procedures are for

Top-Down Design

- Freely call a high-level function whenever you feel you need it and *believe that it is already available and works correctly, even before you've written it*
- Worry about how to write this high-level function later
- May need to do this several times
  - The high-level function is only one level simpler than the problem being solved: it may still not be easy to directly define it in terms of the operations available in the programming language
  - Freely call another slightly lower-level function, assume it exists and works, worry about how to define it later

Why It Makes Things Easier

- Every time you write a part of the code, you only need to think/write something like this

```
reduce(plus, map(square, filter(is_prime, range(2, 100))))
```

- Not this

```
ans = 0
for x in range(2, 100):
    is_prime = True
    for i in range(2, x):
        is_prime = is_prime and x % i != 0
    if is_prime:
        ans += x * x
```

Why It Makes Things Easier

- Defining functions allows you to break the programming process down into separate, manageable levels and *think about the levels one at a time*

Example: Sudoku Checker

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Example: Sudoku Checker

```
def valid(grid): # assume read as list of rows
    return all_rows_valid(grid) \\
       and all_columns_valid(grid) \\
       and all_subgrids_valid(grid)
```

Example: Sudoku Checker

```
def all_rows_valid(grid):
    return all(map(valid_row, grid))
```

Example: Sudoku Checker

```python
def valid_row(row):
    return all(map(in_row, range(1, 10)))
```

Example: Sudoku Checker

```
def valid_row(row):
    def in_row(i):
        return any(elem == i for elem in row)
        # Python magic
        # return i in row
    return all(map(in_row, range(1, 10)))
```

Example: Sudoku Checker

```
def valid_row(row):
    return all(i in row for i in range(1, 10))
```

Example: Sudoku Checker

```
def all_columns_valid(grid):
    return all(valid_column(c) for c in range(0, 9))
```

Example: Sudoku Checker

```
def valid_column(c):
    return all(i in column for i in range(1, 10))
```

Example: Sudoku Checker

```
def valid_column(c):
    column = [grid[r][c] for r in range(0, 9)]
    return all(i in column for i in range(1, 10))
```

Exercises

- Complete `all_subgrids_valid`
- Solve it in C++

How to Code Fast(er)
Part 3: How to Write Loops

What About Loops That Don't Conform to the Patterns?

- There are Z zeros and O ones in a list L, in random order
- You are allowed to swap consecutive elements of L
- Perform swaps on L so that the Z zeros come before the O ones
- Print out the list of pairs of indices to swap

How to Write Loops

- DON’T try to run the loop in your head
- DON’T think about all the steps
- Focus on how to move from *just one step* to the next

How to Write Loops

1. Look at the goal
   - Identify a measure of progress towards that goal – this is the loop counter
   - Think carefully about all the information you need to keep track of – variables
2. **Believe** that the loop *already works before you've written anything*, for steps 1 to $i-1$
3. Using the values *you already have* for $i-1$, figure out a way to make your variables contain the correct values for step i
4. Do the easy parts
   - Figure out the terminating condition
   - Write the initial values of all variables

A Simple Example: Sum of the First n Positive Integers

- Step 1: Look at the goal
  - We want the sum of n numbers
  - Measure of progress: how many numbers added so far, call this i

```
for i in range(:
```

A Simple Example: Sum of the First n Positive Integers

- Step 1: Look at the goal
  - We want the sum of n numbers
  - Measure of progress: how many numbers added so far, call this i
  - Information we need to keep track of: sum so far, call it s

```
s
for i in range(:
```

A Simple Example: Sum of the First n Positive Integers

- Step 2: Believe
  - I’m at step i right now
  - s *already* contains the sum of the first $i - 1$ numbers

```
s
for i in range(:
```

A Simple Example: Sum of the First n Positive Integers

- Step 3: Move from $i-1$ to i
  - To make the sum of the first i numbers, add i to the sum of the first $i-1$ numbers

```
s
for i in range(:
    s += i
```

A Simple Example: Sum of the First n Positive Integers

- Step 4: Easy parts
  - Terminating condition: stop when i exceeds n

```
s
for i in range(, n+1):
    s += i
```

A Simple Example: Sum of the First n Positive Integers

- Step 4: Easy parts
  - Terminating condition: stop when i exceeds n
  - Initial values
    - The sum of 0 numbers is 0
    - Start at $i = 1$

```
s = 0
for i in range(1, n+1):
    s += i
```

Notice

- Code is not written from top to bottom, left to right
- Fill in the details in the order which makes sense to you

Exercise: Try Solving It!

- There are Z zeros and O ones in a list L, in random order
- You are allowed to swap consecutive elements of L
- Perform swaps on L so that the Z zeros come before the O ones
- Print out the list of pairs of indices to swap

Step 1: Look at the Goal

- We need to put Z zeros in front
- Measure of progress: number of zeros we have already placed in front, call it i
- Need to keep track of swaps done so far

```
ans
for i in range(:
```

Step 2: Believe

- At step i, $i - 1$ zeros are already at the front of the list
- `ans` already contains the swaps needed to make that happen

```
ans

for i in range(:
```

Step 3: Move from $i - 1$ to i

- We need to look for a (*single*) zero from the sublist of L starting at index i to its end
- And then repeatedly swap it to the left until it's at position i

```
ans
for i in range(:
  # find a zero from L[i:]
  # “bubble” it towards i
```

Step 4: Easy Parts

```
ans = []
for i in range(0, Z):
  # find a zero from L[i:]
  # “bubble” it towards i
```

Next: Filling in the Details (Remember: Top-Down Design)

- New sub-goal: find a zero in `L[i:]` and save its index to k
- New sub-goal: Repeatedly swap the zero forward until it's at position `i`

```
ans = []
for i in range(0, Z):
  # find a zero from L[i:]
  # “bubble” it towards i
```

Step 1: Look at the Goal

- Look for a zero in `L[i:]`
- Measure of progress: how much of `L[i:]` we've looked at, call it j
- Need to know if we've already seen a zero or not
- Need to know position of the *leftmost* zero, call it k

```
ans = []
for i in range(0, Z):
  has_0, k
  for j in range(:
```

Step 2: Believe

- At step *j*, either…
- A zero exists in `L[i:j]`
  - `has_0` is already set to `True`
  - k already contains the index of the leftmost zero
- There are no zeros in `L[i:j]`
  - `has_0` is `False`
  - k doesn't contain a useful value

```
ans = []
for i in range(0, Z):
  has_0, k
  for j in range(:
```

Step 3: Move from $j-1$ to j

- Case 1: A zero exists in `L[i:j]`
- Variables contain the correct answer for `L[i:j+1]` also
- No need to do anything

```
ans = []
for i in range(0, Z):
  has_0, k
  for j in range(:
    if has_0:
      # do nothing
    else:
```

Step 3: Move from $j-1$ to j

- Case 1: A zero exists in `L[i:j]`
- Variables contain the correct answer for `L[i:j+1]` also
- No need to do anything

```
ans = []
for i in range(0, Z):
  has_0, k
  for j in range(:
    if not has_0:
```

Step 3: Move from $j-1$ to j

- Case 2: No zeros in `L[i:j]`
  - Subcase 2.1: If `L[j]` is 1, there are still no zeros in `L[i:j+1]`
    - No need to do anything

```
ans = []
for i in range(0, Z):
  has_0, k
  for j in range(:
    if not has_0:
      if L[j] == 1:
        # do nothing
      else:
```

Step 3: Move from $j-1$ to j

- Case 2: No zeros in `L[i:j]`
  - Subcase 2.1: If `L[j]` is `1`, there are still no zeros in `L[i:j+1]`
    - No need to do anything
  - Subcase 2.2: If `L[j]` is `0`, there is a zero in `L[i:j+1]` and this zero is the leftmost zero
    - Set `has_0` to `True`
    - Set `k` to `j`

```
ans = []
for i in range(0, Z):
  has_0, k
  for j in range(:
    if not has_0 and L[j] == 0:
      has_0, k = True, j
```

Step 4: Easy Parts

- Termination
  - We have the correct values of the variables for `L[i:j]` at step j
  - When j reaches `len(L)`, we have the correct values for `L[i:]`, which is the problem we originally wanted to solve
- Initial values
  - Start at $j = i$
  - There are no zeroes in `L[i:i]`
  - k can be any garbage value

```
ans = []
for i in range(0, Z):
  has_0, k = False, -1
  for j in range(i, len(L)):
    if not has_0 and L[j] == 0:
      has_0, k = True, j
```

Step 1: Look at the Goal

- Move the ith zero to index i, recording the swaps made
- Measure of progress: where the ith zero is, call it j
  - Because it moves forward, we need to loop in reverse

```
ans = []
for i in range(0, Z):
  has_0, k = False, -1
  for j in range(i, len(L)):
    if not has_0 and L[j] == 0:
      has_0, k = True, j
  for j in range(,,-1):
```

Step 2: Believe

- The ith zero is already at index $j + \mathbf{1}$, and `ans` already holds the swaps required for it to get there from its original position in the list

```
ans = []
for i in range(0, Z):
  has_0, k = False, -1
  for j in range(i, len(L)):
    if not has_0 and L[j] == 0:
      has_0, k = True, j
  for j in range(,,-1):
```

Step 3: Move from $j+1$ to j

- Perform a *single* swap
- Since `ans` already holds the swaps for it to get to $j+1$, record *only one more* swap for it to get to j

```
ans = []
for i in range(0, Z):
  has_0, k = False, -1
  for j in range(i, len(L)):
    if not has_0 and L[j] == 0:
      has_0, k = True, j
  for j in range(,,-1):
    L[j], L[j+1] = L[j+1], L[j]
    ans.append([j, j+1])
```

Step 4: Easy Parts (A Bit Less Easy This Time)

- Termination
  - After step $j = i$, the ith zero is at position i
  - Stop when $j = i - 1$
- Initial values
  - The ith zero is initially at index k
  - No swaps needed to move it from index k to index k
  - Start at index $k - 1$

```
ans = []
for i in range(0, Z):
  has_0, k = False, -1
  for j in range(i, len(L)):
    if not has_0 and L[j] == 0:
      has_0, k = True, j
  for j in range(k-1, i-1, -1):
    L[j], L[j+1] = L[j+1], L[j]
    ans.append([j, j+1])
```

And We’re Done!

- Try it:
  - Assign values to Z and L
  - Print L and ans at the end
- Exercise: can you solve this problem without being given Z in advance?

```
ans = []
for i in range(0, Z):
  has_0, k = False, -1
  for j in range(i, len(L)):
    if not has_0 and L[j] == 0:
      has_0, k = True, j
  for j in range(k-1, i-1, -1):
    L[j], L[j+1] = L[j+1], L[j]
    ans.append([j, j+1])
```

Summary: How Do the Fastest Coders Do It?

1. They don’t think about some parts of the program at all
   - Many parts are “standard” which they’ve seen/done hundreds of times
   - They are common to many programming problems
   - Learn how to write these simpler pieces *without thinking about them*, so that you don’t have to be reinventing them while solving a harder problem

Summary: How Do the Fastest Coders Do It?

2. They don't think about all parts of the program at the same time
    - First, think of a *high-level* way of expressing the program
    - Then, break it down part by part

Summary: How Do the Fastest Coders Do It?

3. They don’t think like a computer, they think like a programmer
   - Don’t follow all the steps of a computation
   - Focus on how to move from one step to the next

Practice Problems

- https://progvar.fun/problemsets/get-started
- https://progvar.fun/problemsets/coding-faster