>>> **a + d**

TypeError: unsupported operand type(s) for +: 'int' and 'str'

- Hmm, not that we really expected the last one to work (how the heck do you add a phrase to a number?), but we're still not really sure what happened. At this point it doesn't really matter, but we do get our first glimpse into the concept of "data types."

- **Data Types**: data always has a "type"
  - We've seen three "types" so far, integers (the int type), strings (the str type), and None (the Nonetype type).
  - An **integer** is any whole number, negative or positive. 5 is an integer. So is -18. But 6.79 is *not* an integer.
  - A **string** is any string of characters within quotes. The plus sign between two strings is special because it "concatenates" them, or just puts them together, not adds them like it does with integers.
  - A **None** is kind of like an empty value and becomes more important later. So far, we know that the **print** function returns a None value though!
  - There is also a **float** type. A floating point number is any number that's not an integer (has a decimal point). So 6.79 is a float. 5.0 is a float. 5 is *not a float*.
  - Examples:

    >>> **a = 5**

    >>> **b = "5"**

    >>> **a + b**

    TypeError: unsupported operand type(s) for +: 'int' and 'str'

    >>> **a + int(b)** #Typecasting b into the int type using int()

    10

    >>> **str(a) + b** #Typecasting a into the str type using str()

    '55'

    >>> **c = "Not a number"**

>>> **type(c)** #The type() function returns the data type

<class 'str'> #The str (string) type

>>> **type(a)**

<class 'int'> #The int (integer) type

>>> **type(b)(type(c)(a)) + b**

'55'

- Here, I've introduced this concept of "typecasting." We can try making a string into an integer and vice versa, and the way we use the "+" can change with the type of data we are using it on. Typecasting changes the "type" of the data so we can do things to different types of data, which will be very useful in the future.
- Let's also look at what happens in this last example a little more closely:

>>> **type(b)(type(c)(a)) + b**

~type(b) evaluates to str; so far: **str(type(c)(a)) + b**

~type(c) evaluates to str; so far: **str(str(a)) + b**

~a evaluates to the integer 5; so far: **str(str(5)) + b**

~str(5) evaluates to the string '5'; so far: **str('5') + b**

~str('5') evaluates to the string '5'; so far: **'5' + b**

~b evaluates to the string '5'; so far: **'5' + '5'**

~'5' + '5' **concatenates** '5' to '5', so we get **'55'** as our final result

- The goal of confusing things like this is to get us to understand how Python interprets and understands the things we put in it. If we have a strong grasp of how Python works, we can build more powerful and very clear programs without having to guess *why* things work the way they do.

- **Operators**: the simplest ways to manipulate and interpret data
  - You need to know the following operators:

    + (addition or concatenation)

    - (subtraction)

***** (multiplication)

/ (general division)

// (integer division)

% (remainder)

- Examples:

  >>> **"3" + "4"** #String concatenation

  '34'

  >>> **3 + 4** #Normal addition

  7

  >>> **3 – 4** #Normal subtraction

  -1

  >>> **3 * 4** #Normal multiplication

  12

  >>> **3 / 4** #Normal division

  0.75

  >>> **10 // 6** # Cuts off everything to the right of the decimal (1.66667)

  1

  >>> **10 % 6** # 6*1 + 4 (remainder of 4)

  4

  >>> **3 % 4** # 4*0 + 3 (remainder of 3)

  3

- You should also know the following operators, which become important in using variables that are of the "**Boolean**" data type. A Boolean variable holds either a "**True**" or "**False**" value.

  == (compares equality of two things, NOT =, which is used for assignment)

  != (not equal)

  >= (greater than or equal to)

  <= (less than or equal to)

> (greater than)

< (less than)

is (compares whether something IS something else)

is not (compares whether something isn't something else)

- <u>Examples:</u>

```
>>> a = 5                       #We assign "a" to 5,
>>> b = 6.0                     #We assign "b" to 6.0
>>> a > b                       #5 is greater than 6.0
False
>>> a <= b                      #5 is less than or equal to 6.0
True
>>> c = type(a) == type(b)      #type(a) is int, type(b) is float; int is equal to float
>>> c
False
>>> c is not True               #False isn't True
True
>>> b = 5.0                     #Reassigned "b" variable to equal the float 5.0
>>> a > b                       #5 is greater than 5.0
False
>>> a == b                      #5 is equal to 5.0
True
>>> a is b                      #5 is the exact same thing as 5.0
False
>>> (a is b) is not (a <= b)    #False isn't True
True
```

- Hmm, this got complicated really fast. Essentially, with these operators, we have to evaluate the expression to see if it's true or false. For

example, let's be Python and evaluate the last expression together.

>>> (a is b) is not (a <= b)

~ **a** evaluates to **5**, **b** evaluates to **5.0**	**(5 is 5.0) is not (5 <= 5.0)**
~ **(5 is 5.0)** evaluates to **False**	**(False) is not (5 <= 5.0)**
~ **(5 <= 5.0)** evaluates to **True**	**(False) is not (True)**

~ **False is not True** evaluates to **True**

To understand this, let's think about it. If I make the statement, "False is not the same thing as True," am I lying? If I am lying, I am making a false statement and python would catch me, returning False! But what I said is actually True! "False isn't True" is a fact, so Python returns True. Confusing, I know, but you'll get the hang of it with more practice. It gets a lot less theoretical with **if statements**, I promise!

- **If statements**: using Booleans to carve out the backbone of computer logic
  - If statements are our first look at "conditionals," which will be ridiculously important for everything we're going to do in this workshop.
  - if (some expression):

            some executable code

    elif (some other expression):

            more executable code

    else:

            kind of the "default" case… Executable code for all other cases

  - My **approach** to conditionals:
    - Step 1: Evaluate the expression to see if it's true
    - Step 2a: If the expression evaluates to True, execute all the code inside the block, then skip any elifs/elses that remain
    - Step 2b: If the expression evaluates to False, you *do not* go inside the block and instead continue to the next block and see if *that* is true
  - You may have multiple elifs but only a single else

- You may do anything within a block… including write more ifs and elifs
- Examples:

```
>>> a = 5
>>> if a % 2 == 0:                #if the number is divisible by 2, it's even
            print("Even")
    else:                         #Otherwise, it's odd
            print("Odd")
Odd
>>> a = 2046         #2 + 0 + 4 + 6 = 12, which is divisible by 3, so 2046 is too
>>> if a % 2 == 1:
            print("Odd")
    else:
            if a % 3 == 0:    #If a number is divisible by 2 and 3, it is by 6 too!
                  print("Multiple of 6!")
            print("Even")
Multiple of 6!
Even
>>> false_ish = True
>>> true_ish = False
>>> if false_ish == true_ish:           #True equals False evaluates to False
            print("Same")
    elif true_ish:                      #False evaluates to False
            print(false_ish)
    elif not false_ish:                 #not True evaluates to False
            print(true_ish)
```

```
else:                    #Default answer, if it didn't go into any of the others
    print("I'm confused")
```

I'm confused

- Okay then, let's explain step-by-step (and sorry for trolling you a little haha). We begin by assigning two variables, false_ish and true_ish. The names are pretty tricky but essentially false_ish is True and true_ish is False. Simple enough so far?

  Now let's look at the "if" block first. False_ish evaluates to True and true_ish evaluates to False. Does True == False? Nope, so we're not going to go inside that block.

  Now let's look at the first elif block. True_ish evaluates to False. Since that expression is False, we don't go inside the block.

  With the second elif block, false_ish evaluates to True and (not True) then evaluates to False. But we need the expression to evaluate to True if we want to go in! So we ignore and move to execute all the code in the else statement, which prints out "I'm confused." Don't worry if you still don't get it, read it over a few times, plug it into Python Tutor, and play around with it in your shell. You'll understand it eventually.
- The really nice thing about if statements is we can write a program that **plans for different things**. For example, let's think about the game we're going to play later called Hog. There are many cases where we roll a dice and *what* we roll changes how we play the game, so we use lots of ifs and elifs and elses to **hand off control** as we move through the program.

- **While Loops**: using iteration to perform actions or computation
  - Iteration is this concept of repetition until we reach some desired "end," and we use loops to perform these iteration computations
  - Loops are our first real intro to Computer Science… Everything up till now has been *Syntax-based*, mostly so we understand how and why Python

works the way it does, but loops now introduce us to logic-based *computation*, a very applicable concept that is very relevant in other languages and in general very useful in Computer Science

- Anatomy of a while loop:

  >>> while (some expression):

  some executable code/computation so we don't get stuck (infinite loop)

- My approach to iteration
  - 1. Evaluate the expression (true, go inside the while loop, false, skip it)
  - 2. While inside the while loop, you need to perform some computation that changes the expression so it is closer to the desired result(this is very general… look at examples for help)
  - Once you're done executing all the code in the block, go to the start of the while loop again and repeat steps 1 and 2.
  - 3. Once you're done with the computation, execute all the code after the while loop normally

- Examples:

```
>>> count = 0
>>> while (count <= 3):
        print(count)
        count += 1                    #Advances count by 1 so we're not stuck
1
2
3
>>> count
4                        #When count became 4, the while expression became
False
>>> starter = 3
>>> while count > 1:                      #We can use the count to figure out
```

3^4

```
        starter = starter*3                    #Performs the computation on our starter
        count -= 1                             #Prevents an infinite loop
        print("Starter:", starter, "…", "Count:", count)
```

Starter: 9 … Count: 3

Starter: 27 … Count: 2

Starter: 81 … Count: 1

>>> count

1

'''Here see that count is down to 1(and made the expression false to stop the while loop). Now you might be wondering…. Why did he make the expression, while count > 1? The reason has to do with starter, since it's already 3, we already have 3^1 and only need to multiply it by 3 three more times'''

>>> starter #We performed some computation on 3 to get it to 81

81

- So at this point we've witnessed how we can use iteration to perform some kind of computation and this will become important when we want to start writing programs that we can use to do "stuff" like some pretty complicated computations
- Let's talk about the Fibonacci sequence now. The Fibonacci sequence is a sequence that starts with two numbers, 0 and 1. From there, the rest of the sequence is defined by making the next term the sum of the two preceding terms.

  The first 10 terms are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

- Very difficult Fibonacci Example:

```
>>> current, next = 0, 1                    #We start with these two as "givens"
>>> nth_term = 1
>>> while nth_term <= 10
        print(current)
        current, next = next, current + next
        nth_term += 1
        '''Alternatively, if you don't understand this simultaneous
assignment:
        temporary = current                 #We temporarily need to store the
value
        current = next                      #We reassign current to be the new
current
        next = next + temporary             #Next term is the sum of the
preceding two
```

  The answer should print 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 all on their own lines

- It truly is okay if you don't understand this fib sequence yet. It's a very limited amount of time to learn a tremendous amount of CS, so I don't expect you to master the hard stuff. If you get stuck, put it in Python Tutor to figure it out!

- **def statements:** using functions to "modularize" our programs and allow for different computations to be done on different arguments
  - Functions will be our first introduction to this idea of **"functional abstraction"** which is just a really fancy way of saying that we want to be able to **change** our programs very easily so we make a bunch of *small* parts that work together to accomplish a goal. If we were to make the program all just one huge block of code, it would be *very* complex and *very* difficult to change in the future, so we use lots of functions (which

each have their own blocks of code) to do different things that work together.

- In terms of working on projects, functional abstraction is also a really important idea because it helps us work together with other people. There are many ways to write code that do the same thing—with abstraction, we don't really *care* how someone else's code works, just that it *does*, and we assume it works the way it needs to. Let's take a look at some examples so this isn't as theoretical.
- Anatomy of a def statement:

  def function_name(argument1, argument2):

  some executable code

  return statement (ends execution of code in function… default is None)
- Examples:

  >>> **add(1, 5)** #Built in function add() for addition

  6

  >>> **sub(3, 2)** #Built in function sub() for subtraction

  1

  >>> **mul(48, .5)** #Built in function mul() for multiplication

  24

  So in these three examples above, we begin to see how functions work. There is a call expression, with the function's name, and arguments within parentheses that the function is applied to. Remember print? That's a built in function too. Other important functions you should know include the max() and min() functions which find the maximum/minimum of a set of numbers

  >>> **max(1, 2, 3, 4, 5, -5)** #Find the biggest number in this data set

  5

  >>> **min(1, 2, 3, 4, 5, -5)** #Find the smallest number in this data set

-5

```
>>> add(5, max(2, 6), 3, mul(10, 2), min(1,1))          #5 + 6 + 3 + 20 + 1
35
```

Let's explore how python works to interpret an expression like the one above. How do we get to the result we end up getting to? Well, first there is a function call to the built-in function, "add," which we know will add up all of the arguments. Okay, so 5 comes in as an argument and evaluates to 5, but then we get to max(2, 6)… We can't "add" that in the conventional sense, so we first evaluate it. Max(2, 6) calls the function "max" which looks at its arguments, 2 and 6, to find which is the biggest. Since 6 is the biggest argument, max(2, 6) evaluates to 6. 3 evaluates to 3, mul(10, 2) evaluates to 20, and the minimum of a data set will always be the smallest number present, so min(1,1) evaluates to 1.

These functions are built-in, so we don't know how they are coded, but let's write these functions using what *we've* already learned.

```
>>> def addition(x,y):
        return x + y            #This is what our function will
evaluate to
>>> def subtraction(x,y):
        return x – y
>>> def multiplication(x,y):
        total = 0
        while y > 0:
            total += x          #We'll add 'x' to total 'y' number of
times
            y -= 1
        return total
```

The way I've coded multiplication(x,y) is a solid and interesting way to

compute multiplication (think about this logically and you'll understand what's happening inside the while loop), but there are still some flaws. Can you see them? Like what happens when I make y a negative number? Does this work when y is 0? How can I fix it for the first flaw? (yes it works when y is 0)

There are other ways that functions can be useful too. What if we want to see if the number is prime? Or what the square of a number is? Let's define these functions too. A number is prime if it is divisible by only 1 and itself (by definition, 1 isn't prime), and a square of a number is simply the number multiplied by itself

```
>>> def is_prime(x):
        count = 2           #The first number we check (more than 1)
        while count < x:
            if x % count == 0:          #If x is divisible by count, go inside
                return False            #Exits the function, returning False
            count += 1
        return True
```

In this function we take a very simple stance and attempt to prove whether it is true or not. Here's the logic of the code in very simple English—we will take all the numbers between 2 and x and see if x is divisible by any of them. If x *is* divisible, it doesn't satisfy the definition of a prime number (divisible only by 1 and itself) and we return False if we find that to be the case. However, if we check all the numbers between 2 and x and x isn't divisible by any of them, we will return True, because it satisfies the definition of a prime number.

```
>>> def square(x):
```

```
        return multiplication(x, x)                #Definition of a square
```

So what have we done here? We used a function we've already defined (our equivalent of the mul built-in function, to multiply x by x and return it. Of course we could also do this simply with the expression (x * x)

- <u>Hard Fibonacci Example (Function):</u>

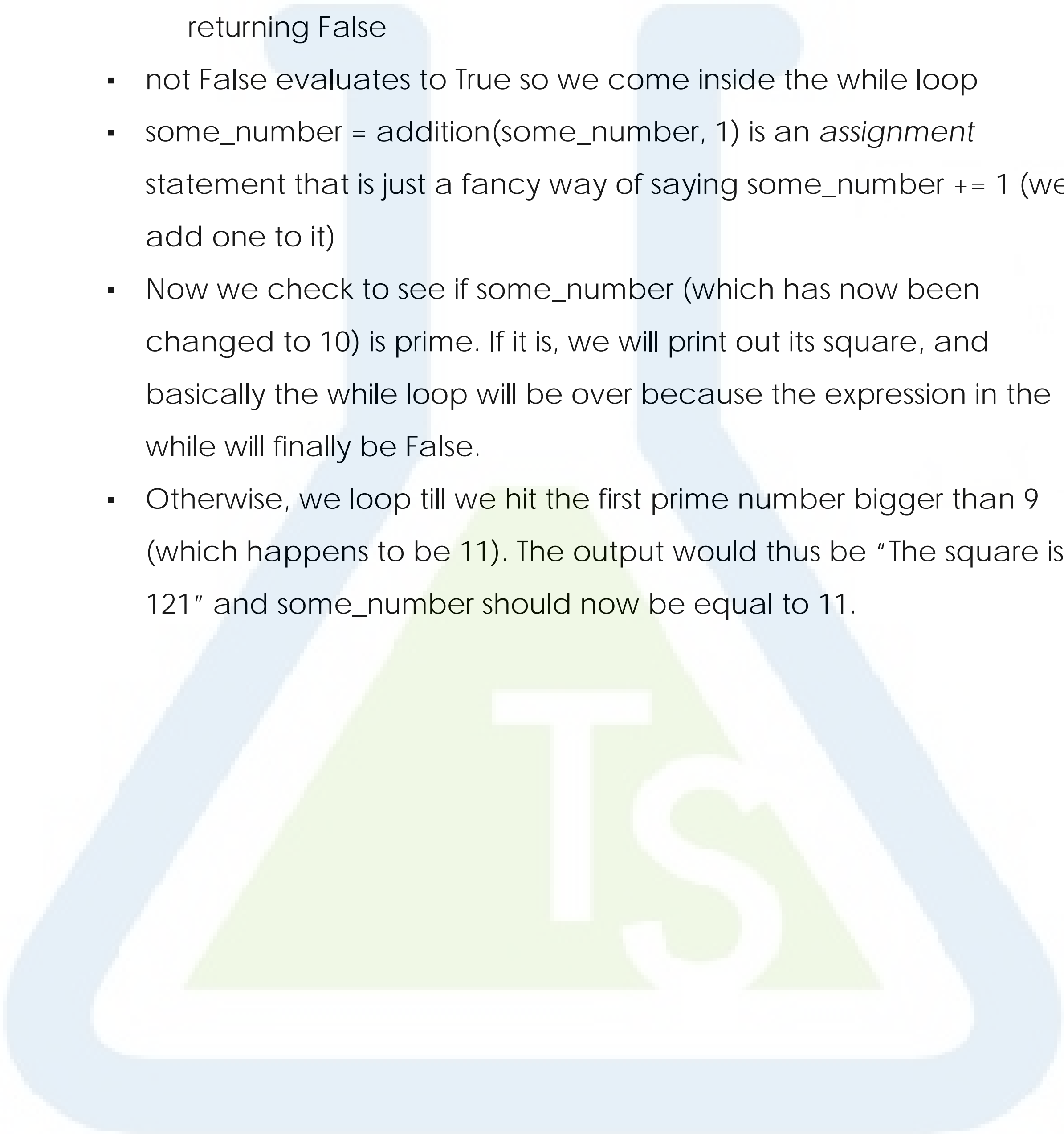
```
>>> def fib(n):                              #Get the nth fibonacci number
        current, next = 0, 1
        if n == 2:
            return next
        while n > 0:
            current, next = next, (current + next)
            n -= 1
        return current
```

- Now we'll use the functions we've defined (this is what you'll be doing in Homework, Labs, Discussion, and our project, Hog, which is what we're working towards).

```
>>> some_number = 9
>>> while not is_prime(some_number):
            some_number = addition(some_number, 1)
            if is_prime(some_number):
                print("The square is " + str(square(some_number)))
```

So let's understand what happened in this example above, and what I'm trying to do with this while loop, line by line.

  - some_number = 9 is an *assignment* statement, assigning 9 to a variable
  - while not is_prime(some_number): defines the expression for which the while loop will run. So we evaluate not is_prime(some_number).



    - Is_prime is a function call and the argument coming in is the int, 9
    - Is_prime takes a look at 9 (using the code above) and determines it is NOT a prime number because it's divisible by 3, returning False
- not False evaluates to True so we come inside the while loop
- some_number = addition(some_number, 1) is an *assignment* statement that is just a fancy way of saying some_number += 1 (we add one to it)
- Now we check to see if some_number (which has now been changed to 10) is prime. If it is, we will print out its square, and basically the while loop will be over because the expression in the while will finally be False.
- Otherwise, we loop till we hit the first prime number bigger than 9 (which happens to be 11). The output would thus be "The square is 121" and some_number should now be equal to 11.

Homework/Lab/Discussion Tips

Computer Science is all about practice, so the more you practice, the more you'll understand how so much of this works.

Some Mini Tutorials if you get stuck:

- Discussion 1:

  is_prime (n): is_prime is the NAME of the function, and n is an inputted argument. We have to figure out if a number n is prime. The definition of prime is that a number is ONLY divisible by itself and 1. Approach? Look at every number less than n and see if n is divisible by it. If any are, the number is NOT prime. Now translate my English to code.
- square_ints(n) and double_ints(n):

  The idea behind these functions is that we are being asked to start at 1 and go till n. What you should recognize here is that we often solve these kinds of problems with iteration and looping.
- Quick loop syntax refresher:

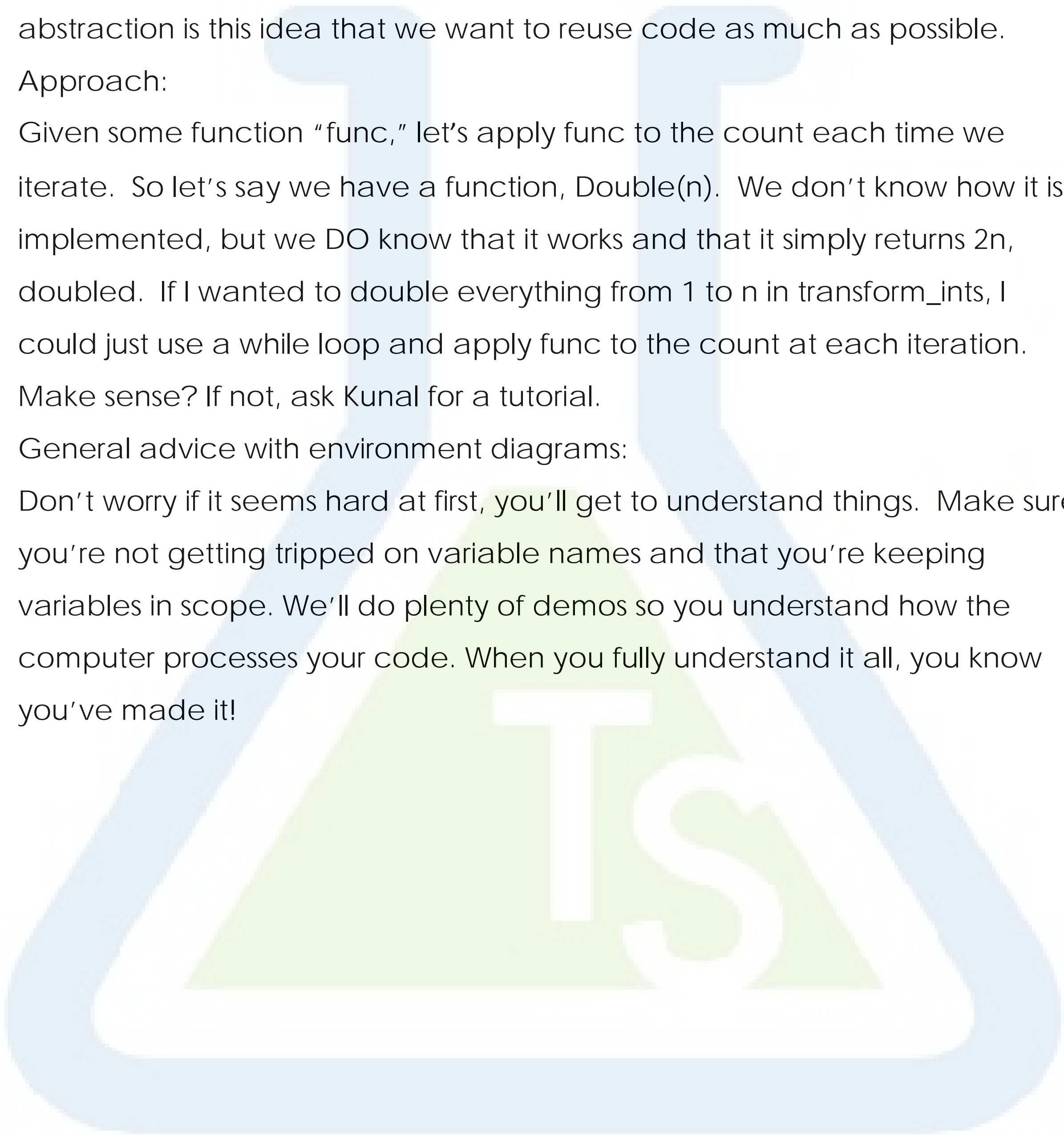
  count = someNumber

  while (conditional involving count):

  do stuff here

  count += 1
- transform_ints(func, n):

  What is this function trying to do? Well we saw with square_ints and double_ints that our code actually looks pretty similar in both cases, right? We just have a while loop incrementing from 1 to n and every time we increment count, we print out a result of the operation we just performed. For double_ints(n), we doubled the count. For square_ints(n), we squared the count. But each time, the code was actually really similar.



In real CS, copy-pasting code is horrible, especially when you get into huge projects. One of my projects last semester involved writing 1500 lines of code to get something to work from scratch, and it was really cool, but only possible due to something called functional abstraction. Functional abstraction is this idea that we want to reuse code as much as possible.

- Approach:
  Given some function "func," let's apply func to the count each time we iterate. So let's say we have a function, Double(n). We don't know how it is implemented, but we DO know that it works and that it simply returns 2n, doubled. If I wanted to double everything from 1 to n in transform_ints, I could just use a while loop and apply func to the count at each iteration. Make sense? If not, ask Kunal for a tutorial.
- General advice with environment diagrams:
  Don't worry if it seems hard at first, you'll get to understand things. Make sure you're not getting tripped on variable names and that you're keeping variables in scope. We'll do plenty of demos so you understand how the computer processes your code. When you fully understand it all, you know you've made it!

Debugging and Testing Tips (Q & A)

Q: What do I do if I'm failing tests? How do I fix that?

A: Debugging is a tremendous skill in CS. My advice is to write a ton of comments in your code so it makes it really easy to understand when you're going through. One way to debug is using Python Tutor and/or Environment Diagrams to understand what's going on inside your code. Sometimes just missing a parentheses can make code go horribly bad, so make sure you're paying attention and diligently commenting. It also makes it easier for me to debug whenever I go through your code (if you absolutely can't figure something out), because I understand your thought process much more clearly

Q: What is testing? Why is it important?

A: Testing is how we make sure things work. In addition, I've found that when I write tests for something, I'm really forced to consider all the possible "edge cases" (things that can make a program break) and make sure my code doesn't break when those edge cases come up.

Q: But seriously, is it that important?

A: Well let's put it this way: in some fields of CS, you write tests even *before* you write code. This is called Test Driven Development (TDD) and is cool because you know all the things you *can* break before you even start. When you do start, it's easier to not break anything at all. Some people spend their entire lives doing this kind of stuff.

Q: How do I write tests?

A: Included with every autograder and skeleton file is a series of doctests. These doctests give you an example of what the function is supposed to do for it to be correct. It's a really good habit to take a look at these doctests, understand what they're doing, understand *why* they're included, and understand how *you* can expand upon them.

```
someFunc(n1, n2)      #Some function that takes two numbers n1, n2
```

For example, the given doctests might test when n1 and n2 are both positive numbers, but edge cases could be when you have a positive and negative, two negatives, a negative and zero, or a positive and zero.