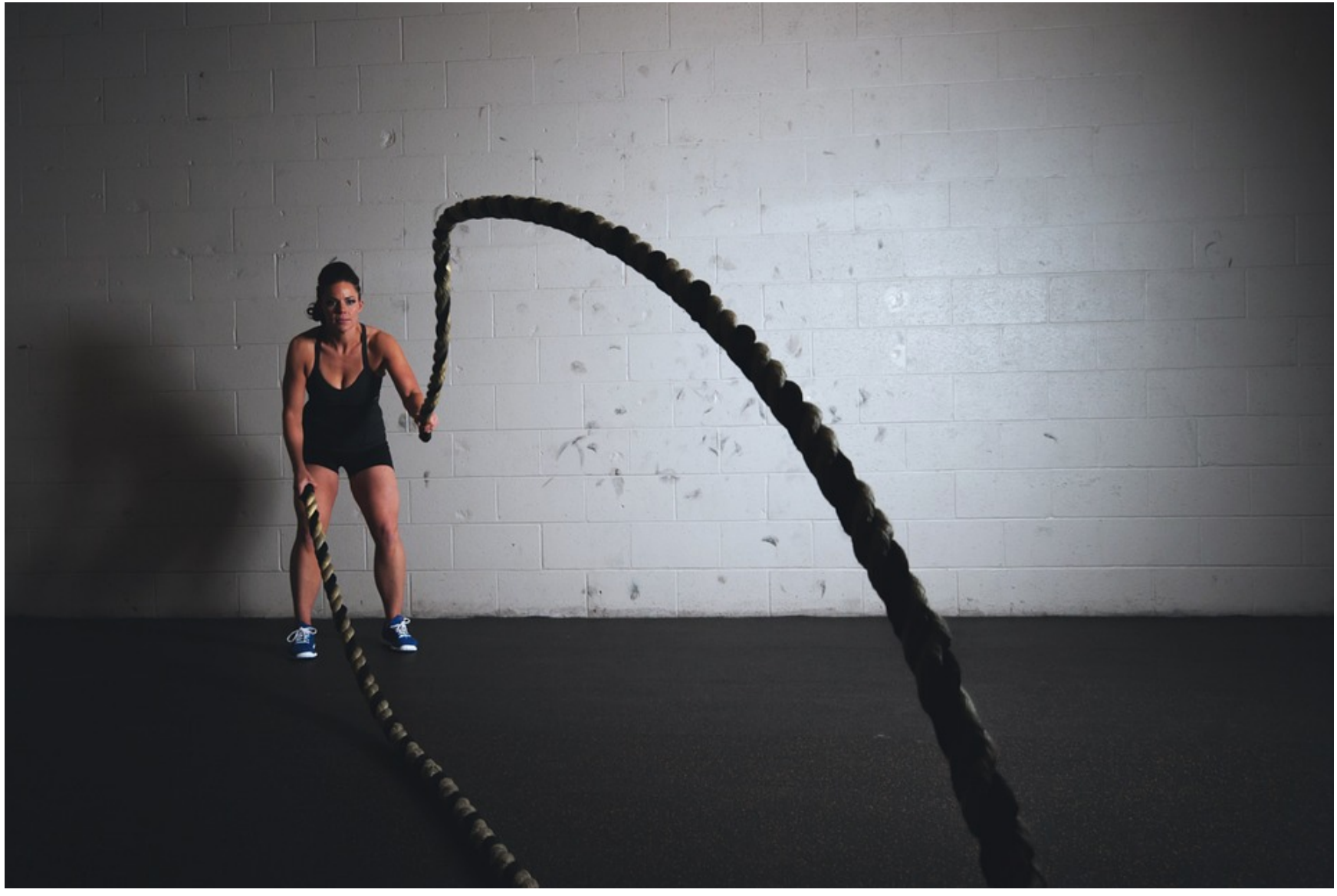
String Exercises



We'll start out with a short DNA sequence as our example string.

Don't try to copy the one here. Make your own sequence, but make sure you only use A, C, G, or T, and make sure it's capitalized.

```
dna = 'ACTAGCTACGCTCGATACGCATCG'
```

Let's check out the type of this variable, just to be sure.

```
print(type(dna))
```

```
<class 'str'>
```

Now let's try running another **function** on the variable. Here's a good example of something that computers are good at: counting.

```
print(len(dna))
```

```
24
```

Now let's see what kinds of different ways we can change this string.
The first one we'll try is to convert it to lowercase. We'll use the lower() **method**. Notice how this **method** is different than **functions** that we've used previously.

```
print(dna.lower())
```

```
actagctacgctcgatacgcatcg
```

Also, it's important to point out that the "dna" variable did not change by applying that method.

```
print(dna)
```

```
ACTAGCTACGCTCGATACGCATCG
```

To truly change a string variable, you need to re-save those changes back to the same variable name.

```
dna = dna.lower()
print(dna)
```

```
actagctacgctcgatacgcatcg
```

Notice that the lower() method just used an empty parenthesis. We'll now use the replace() method to show how to pass **parameters** that tell the method what to do. To demonstrate this, we'll tell Python to replace the "thymine" bases with "uracil" to convert it to an rna sequence.

```
rna = dna.replace('t', 'u')
print(rna)
```

```
acuagcuacgcucgauacgcaucg
```

Another very useful action that Python can perform is called **slicing**. Say we realized that we only wanted to use the first 10 bases of the dna sequence for some analysis. We use **brackets[]** to do this, and then we tell Python where to start and where to end. It's important to note that **counting in Python starts with 0**.

```
print(dna[0:10])
```

```
actagctacg
```

In fact, if we're just starting with the beginning of the string, we don't need to write out the 0.

```
print(dna[:10])
```

```
actagctacg
```

Now, what if a colleague tells us about a 5-bp stretch that we were forgetting from our sequence? We can add that to our original sequence, using the **+ operator**. Remember from last class, that the "+" is used to add numbers, but it can add strings too?? This is an example of **operator overloading**, where an operator can be programmed to do different operations based on the type of object it's working with.

```
missing_stretch = 'agtca'
dna = dna + missing_stretch
print(dna)
print(len(dna))
```

```
actagctacgctcgatacgcatcgagtca
29
```

Another example of operator overloading in strings is the * operator. The * will "multiply" a string howevery many times we tell it. Here we make this new sequence into a repeat.

```
repeat_dna = dna * 2
print(repeat_dna)
print(len(repeat_dna))
```

```
actagctacgctcgatacgcatcgagtcaactagctacgctcgatacgcatcgagtca
58
```

Conditionals

Let's switch back to talking about our Week 1 homework assignment. In Paul's email, we learned that you have to convert the raw_input to an int for the addition to work correctly.

Let's see what happens when we try to convert our *dna* variable to an int.

```
print(int(dna))
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recer

<ipython-input-12-3a6735f6a4be> in <module>()
----> 1 print(int(dna))


ValueError: invalid literal for int() with base 10: 'actagctacg
```

Uh oh, that's not good. It looks like Python is choking when it tries to convert a string of DNA bases to an integer.

We need a way to test if our raw_input (which by default is a string) is the type of a string that can be converted to an integer. Luckily, there is a string method that checks this for us.

```
print(dna.isnumeric())
```

```
False
```

Let's test this on a test number string to make sure it's working like we think it should.

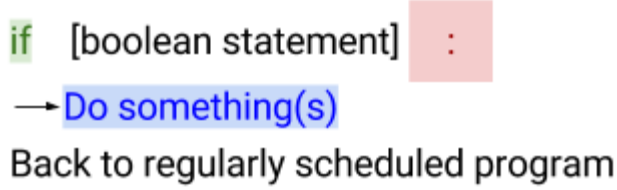
```
test_variable = "42"

print(test_variable.isnumeric())
```

```
True
```

Good. Now we're going to need to use Python **conditional** statements to actually use this new method.

The way to write a conditional statement in Python is like this:




```
if test_variable.isnumeric():
    test_variable_int = int(test_variable)
    print(test_variable_int + test_variable_int)
```

```
84
```

But what if *test_variable* is not a numeric string?

```
test_variable = "agtc"
if test_variable.isnumeric():
    test_variable_int = int(test_variable)
    print(test_variable_int + test_variable_int)
```

Nothing happens. We need a way to let the user know that they didn't follow the rules. That's where **else** comes in.

```
test_variable = "agtc"
if test_variable.isnumeric():
    test_variable_int = int(test_variable)
    print(test_variable_int + test_variable_int)
else:
    print("Your input was not valid.")
```

```
Your input was not valid.
```