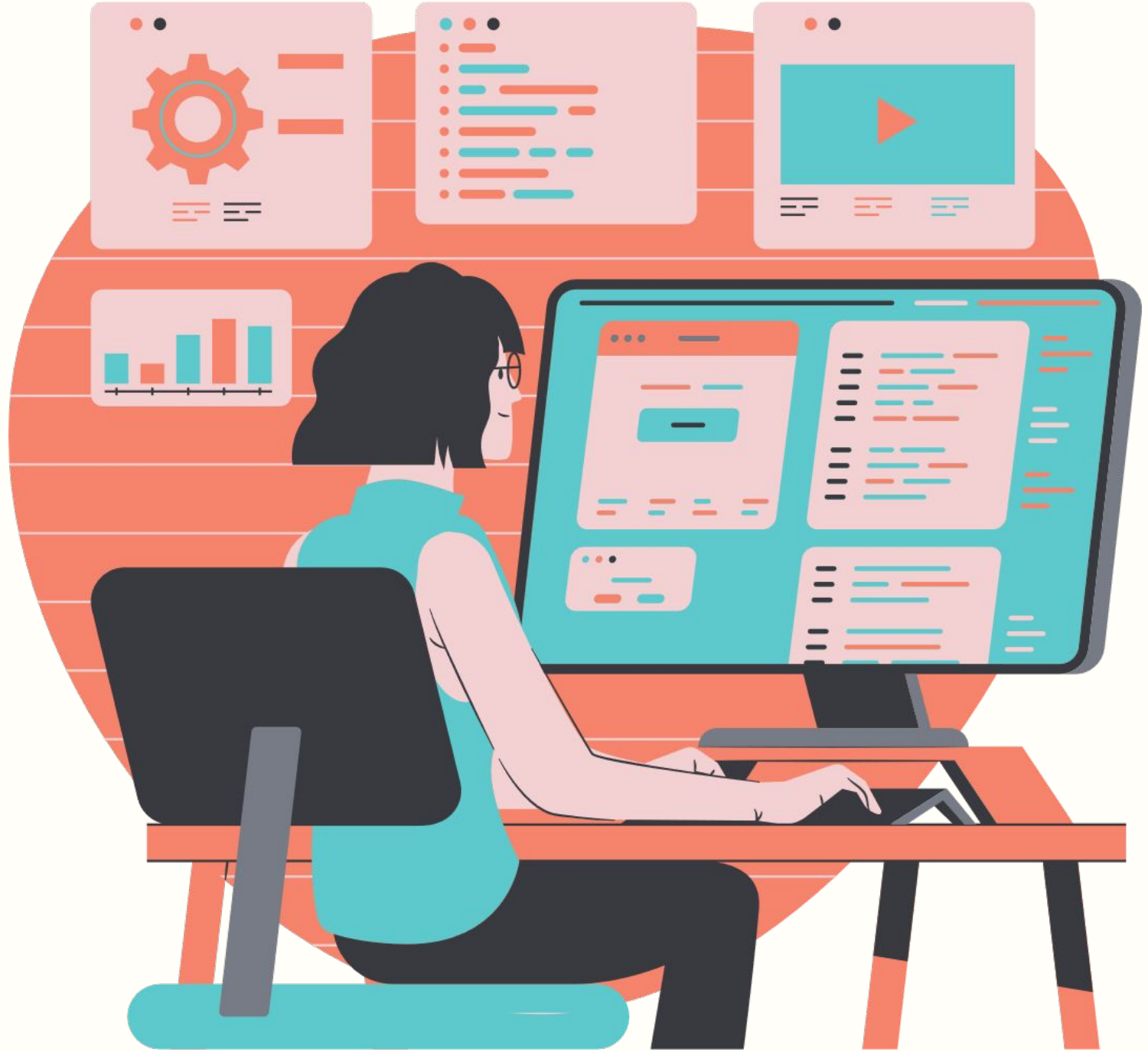





REC 2: LOOPS & SEQUENCES

CIS 1100 (PYTHON), 9.23/4



A set of seven horizontal bars of varying lengths and colors (gray, teal, black, orange, orange, teal, red) arranged in a staggered, overlapping pattern.

LOGISTICS & REMINDERS

1. **HW2: Personality Quiz** is due **09/25 @ 11:59 PM**
 2. **Mandatory code review sign-ups due 09/25 @ 11:59 PM**
 - a. Check your email for details!
 - b. Will be held 09/26–10/02
 3. Optional **Sunday Review Sessions** every Sunday!!
 - a. Check Ed Discussion for updates on the time
 - b. Get extra practice with exam questions
 - c. Ask any conceptual questions 1:1
- 
- A set of seven horizontal bars of varying lengths and colors (gray, teal, black, orange, orange, teal, red) arranged in a staggered, overlapping pattern, mirroring the top-left decorative element.



CLARIFYING FILE HEADERS

- **Name:** Hopefully self-explanatory
- **PennKey:** The one with letters, *not* the 8-digit number
- **Execution:** What you type into the terminal to run your code (including command line arguments)

```
python [filename].py [argument 1] [argument 2] [...]
```

Note: If there are multiple options for an argument, you can give specific examples for each or list all the options in one line (ultimately, if it's clear how to run your program, you'll get credit)
- **Description:** A brief explanation of what your program does at a high level

Let's go through an example together for the upcoming homework!



TABLE OF CONTENTS



01

SEQUENCES

04

WORKSHEET

02

COMMAND LINE ARGS

03

LOOPS

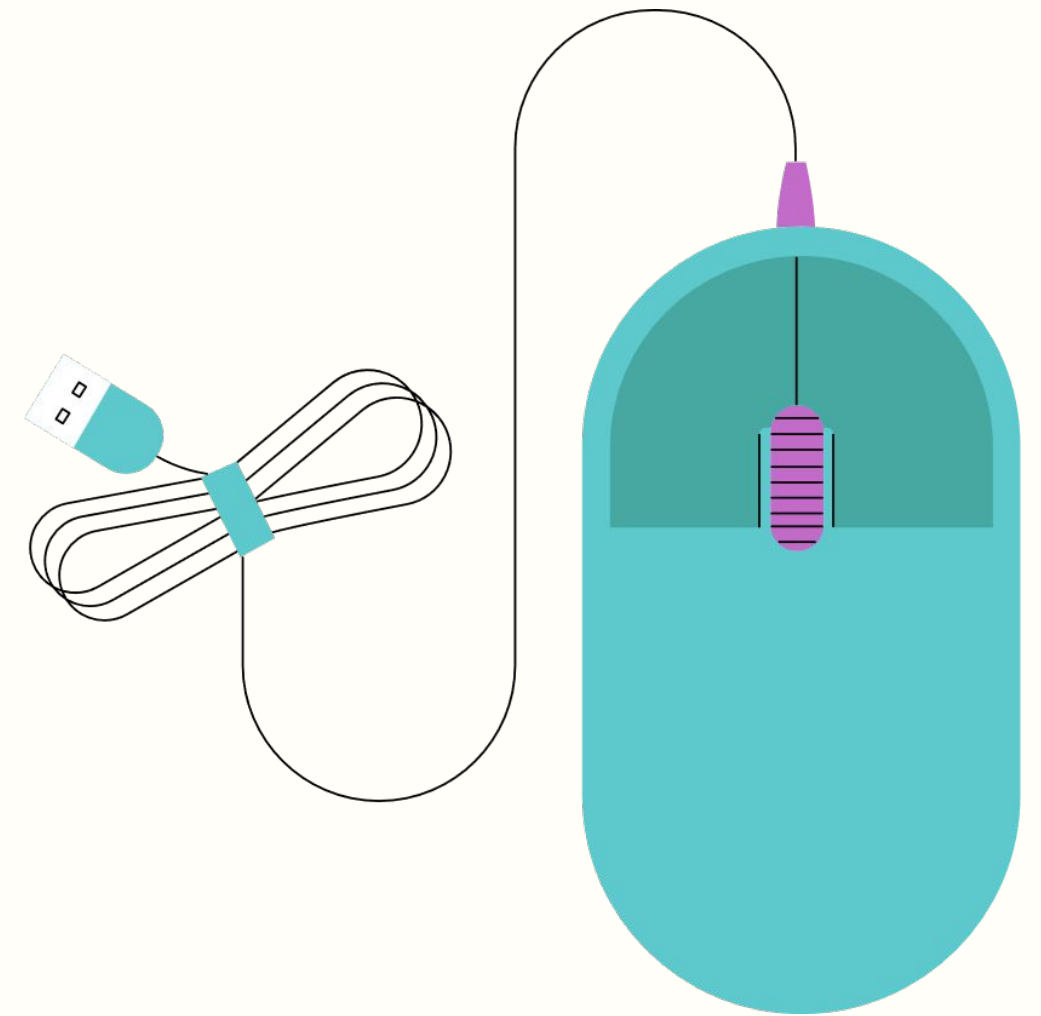




01

SEQUENCES

What are sequences, what types of sequences are there, and how can we operate on them?





SO, WHAT EVEN IS A SEQUENCE?

So far, we've talked about individual values: ints, bools, etc.

Sequences allow us to handle multiple values at once. Specifically:

- Sequences are **ordered** collections of values
- Each element has an index, going from 0 to the length of the sequence



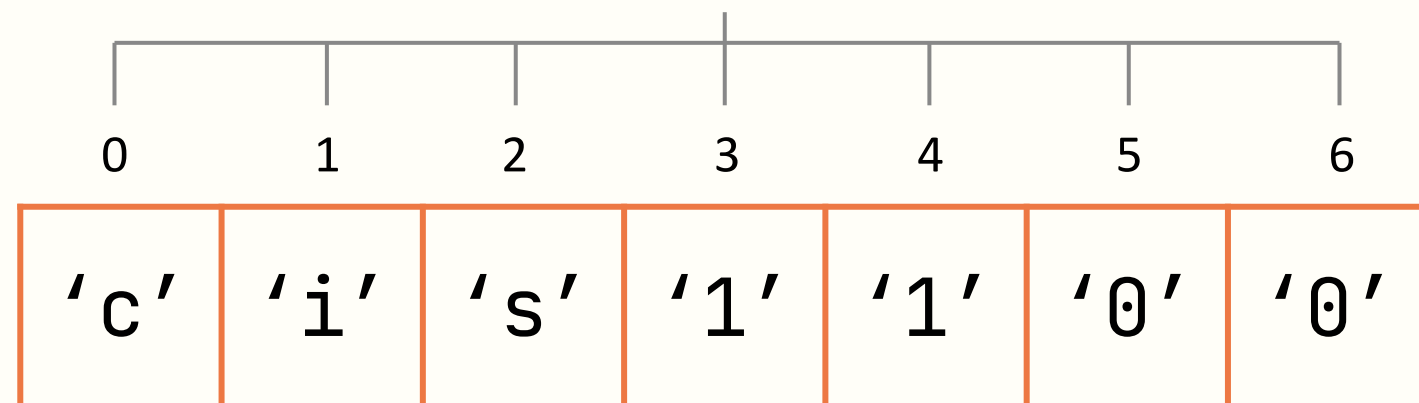


SEQUENCES IN REAL LIFE

- 7-day pill box (each box contains certain pills)
- Spreadsheet column
- Vectors in math
- Strings! (They're sequences of individual characters)



'cis1100'





SEQUENCES IN PYTHON

Type	Element type	Mutable	Example
str	Characters (length-1 strings)	False	"Hello", "cis1100"
list	Anything	True	["a", "b"], [1, False]
tuple	Anything	False	("a", "b"), (1, False)
range	Ints	False	range(1, 3) →



SEQUENCE OPERATIONS

Action	Code	Example
Get the i-th element	<code>seq[i]</code>	<code>"Hello"[2]</code> # = <code>"l"</code>
Get the length of a sequence	<code>len(seq)</code>	<code>len([1, 2, 3])</code> # = <code>3</code>
Concatenate (join together) two sequences	<code>seq1 + seq2</code>	<code>(1, 2) + ('a', 'b')</code> # = <code>(1, 2, 'a', 'b')</code>
Modify the i-th element (Note: This is <i>only</i> for lists)	<code>seq[i] = ___</code>	<code>seq = [1, 2]</code> <code>seq[1] = 3</code> # <code>seq = [1, 3]</code>
Enumerate the indices of each element in the sequence	<code>enumerate(seq)</code>	<code>seq = ['a', 'b']</code> <code>enumerate(seq) = [(0, 'a'), (1, 'b')]</code>



TUPLES VS. LISTS

At first blush, tuples and lists seem very similar—the key difference is that lists are mutable, while tuples aren't. This gives rise to several functions that **only** lists can use:

- `li.append(x)`: Adds element `x` to the end of `li`
- `li.insert(i, x)` : Inserts element `x` to the `i`-th index of `li`
- `li.extend(li2)`: Concatenates `li` with another sequence `li2`





RANGES

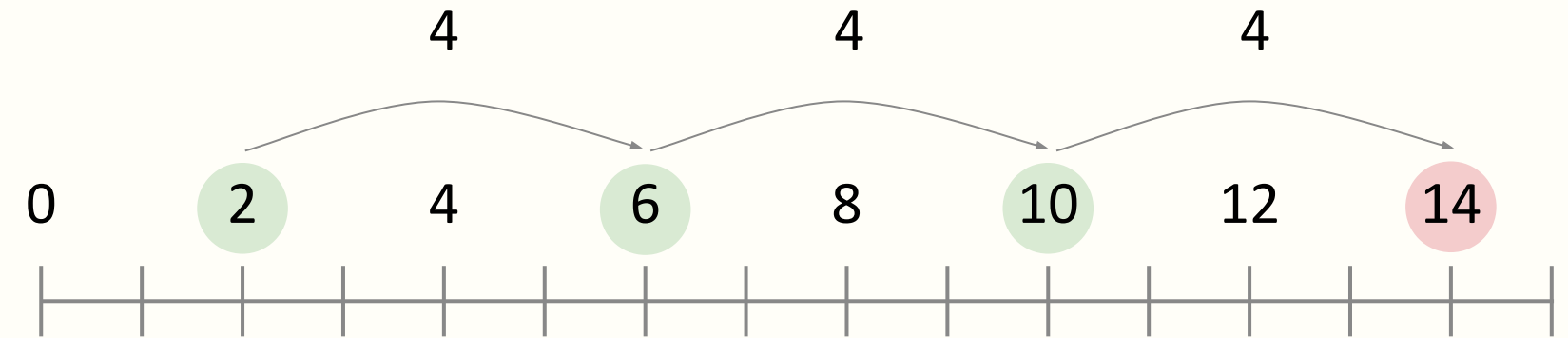
Ranges are a bit different from other sequences in that you don't write out all their values. Instead, they're defined by 3 numbers: the start, end, and step size.

```
range(start, stop, step_size)
```

There are some shortcuts for common steps too:

- `range(start, stop) = range(start, stop, 1)`
- `range(stop) = range(0, stop, 1)`

```
range(2, 14, 4)
↳ 2, 6, 10
```



Quick check-in: What are the values in the following ranges?

- `range(-1, 5)`
- `range(3)`
- `range(4, -1, -1)`





SEQUENCE SLICING



What if we want to get a whole section of a sequence, like the first 5 elements? A subsequence like this is called a **slice**, and Python has a neat shortcut for getting it:

```
seq[start:end:step_size]
```

- This works *almost* the same way as ranges!
- Some shortcuts:
 - `seq[start:end] = seq[start:end:1]`
 - `seq[:end:step_size] = seq[0:end:step_size]`
(similarly, `seq[:end]`)
 - `seq[start::step_size] = seq[start:len(seq):step_size]`
(similarly, `seq[start:]`)
 - ... etc.

Let's try it out: Let

```
s = "I love CIS 1100"
```

What are the following values:

- `s[:3]`
- `s[4:5]`
- `s[5:5]`
- `s[::-1]`





MEMBERSHIP

One final tool that's useful is membership checking. The `in` operator tells us if an element is anywhere in a sequence. For example, with `s = [1, 2, 3, 4] ...`

- `2 in s` → `True`
- `0 in s` → `False`
- `1 in s[1:]` → `False`

With strings **only**, you can even check if substrings are in the sequence. With `s = 'coding' ...`

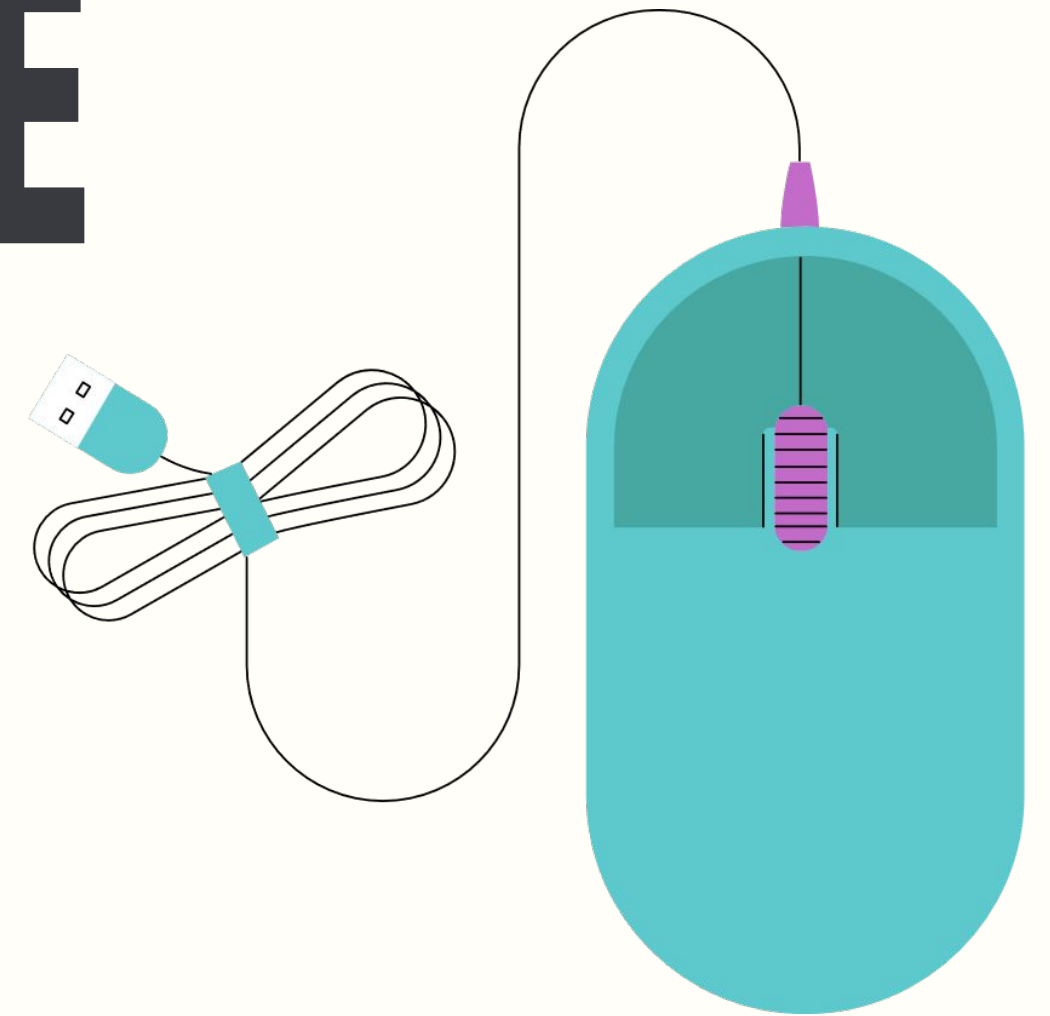
- `'c' in s` → `True`
- `'ing' in s` → `True`
- `'ods' in s` → `False`



02

COMMAND LINE ARGUMENTS

How to get user inputs when you run a program.





WAIT, DON'T WE HAVE INPUTO?

In our first lecture, we showed you how to use `input()` to get user inputs. Sometimes, though, we don't want to wait for the user. We want to know the inputs, right when we run the file.

That is, we want something like this:

```
python filename.py [input 1] [input 2] [... and so on]
```





READING IN INPUTS

Python actually has a way to read in those inputs as a list called `argv`!

```
import sys                # Import the relevant library

inputs = sys.argv        # Use the argv list
```

The first element (`argv[0]`) is always the filename. The rest of the list comprises the inputs **as strings** (separated by spaces).

Example: Let's say you have a file called `inputs.py`. What is `argv` if you run

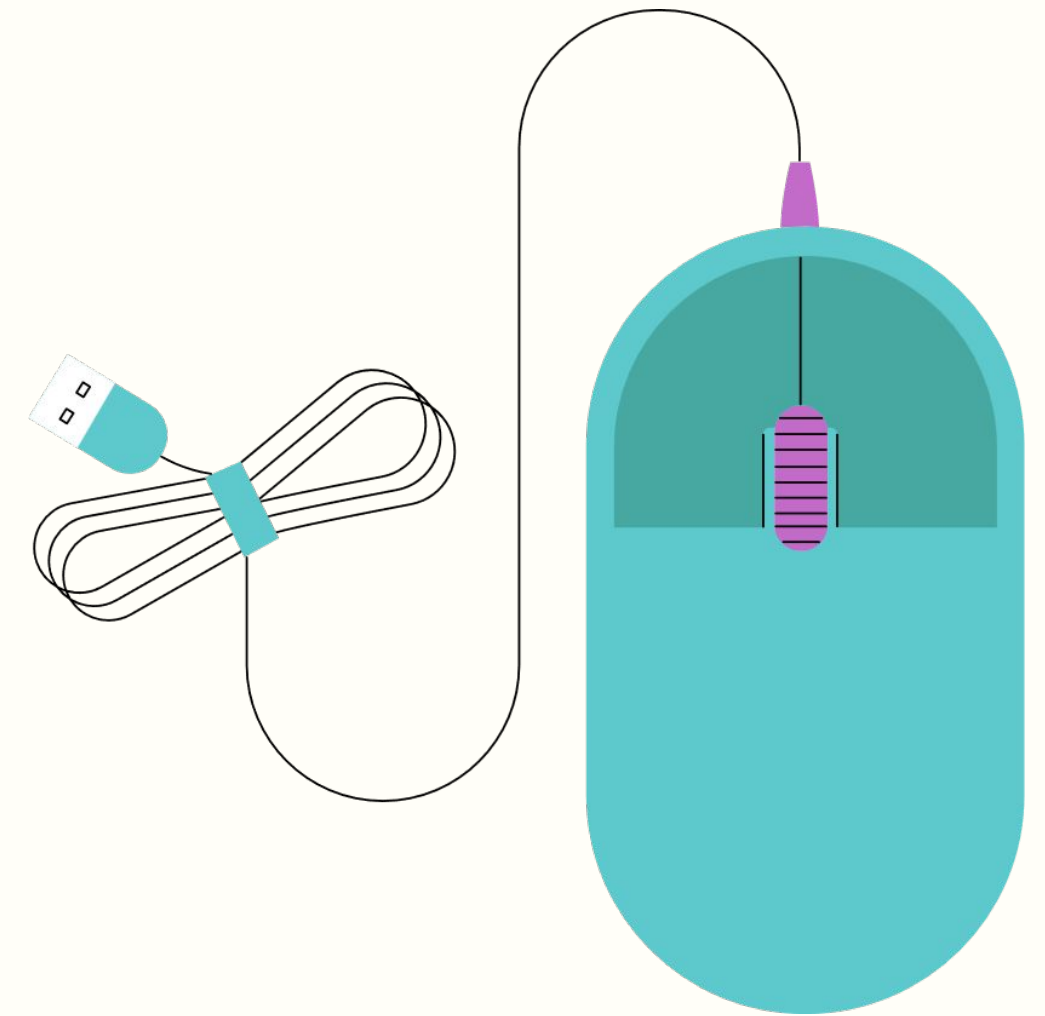
```
python inputs.py 1 A True
```




03

LOOPS

Repeated actions, and sequence iteration





BUT CAN'T I JUST COPY-PASTE?

Let's say I want to print out the numbers 0 to 10. Here's one way:

```
print(0)
print(1)
print(2)
...
print(10)
```

But this takes 11 lines, and I'm lazy (you should be too!). What if there were a way to do it in 2 lines ...





THE FOR LOOP

The for loop goes through each item in an **iterable** (\approx sequence, for now) one at a time, and does a piece of code for each item.

```
for item in seq:  
    # Do something, where "item" is a variable you can use
```

Let's convert the code from the previous slide into a for loop!

Answer:

```
for i in range(0, 11):  
    print(i)
```





SOME USEFUL FOR LOOPS



Do something n times:

```
for _ in range(n):  
    # Do something
```

Go through each character in a string s:

```
for c in s:  
    # Do something
```





A COMMON USE CASE: FILTERS

Let's say you have a list users of tuples (name, age), and you want to just get those that are at least 18. Store these users in a new list. How would you do this? This process, **filtering** sequences, is a common use case for loops!

Hint: Use both loops and conditionals!



Answer:

```
adults = []
for user in users:
    if user[1] ≥ 18:
        adults.append(user)
```





ANOTHER CASE: AGGREGATION

Let's say you have a list of ints, and you want (for whatever reason) to find the largest even element. This kind of problem—where you take a list and output one value—is called **aggregation**, and is commonly solved with **accumulator variables**.

```
nums = ...  
# Accumulator variable  
largest = float('-inf')  
for n in nums:  
    if n > largest:  
        largest = n
```

Here's the code for finding the largest element (not necessarily even). Can you use this to solve our problem?





AND ONE MORE: MAPPING

Let's consider one last scenario: Say you have a list of temperatures in Celsius, and you want to convert them all to Fahrenheit, storing the result in a new list. This kind of problem—where you change each element somehow—is called **mapping**.

Try it out, using what you know about assigning values for lists! Tip: $F = C * 9 / 5 + 32$

Answer:

```
c = ...  
f = []  
for temp in c:  
    f.append(temp * 9 / 5 + 32)
```





LIST COMPREHENSIONS

Python has some syntactic sugar (looks nicer, but is functionally pretty similar) for some of those common operations called **list comprehensions**.

Use Case	Normal For Loop	Comprehension
Filtering	<pre>new_list = [] for elem in seq: if condition: new_list.append(elem)</pre>	<pre>new_list = [elem for elem in seq if condition]</pre>
Mapping	<pre>new_list = [] for elem in seq: new_list.append(new_elem)</pre>	<pre>new_list = [new_elem for elem in seq]</pre>



SOME EXAMPLES

Normal For Loop	Comprehension
<pre>adults = [] for user in users: if user[1] ≥ 18: adults.append(user)</pre>	<pre>adults = [user for user in users if user[1] ≥ 18] or adults = [(name, age) for name, age in users if age ≥ 18]</pre>
<pre>c = ... f = [] for temp in c: f.append(temp * 9 / 5 + 32)</pre>	<pre>f = [temp * 9 / 5 + 32 for temp in c]</pre>



YOUR TURN!

01

Given a list of non-empty string names, return a list of the *indices* of names that start with the letter 'A' (including capitalization). For example, ['Harry', 'Adi', 'Travis', 'Annie'] → [1, 3].

02

Rewrite the following snippet using list comprehensions:

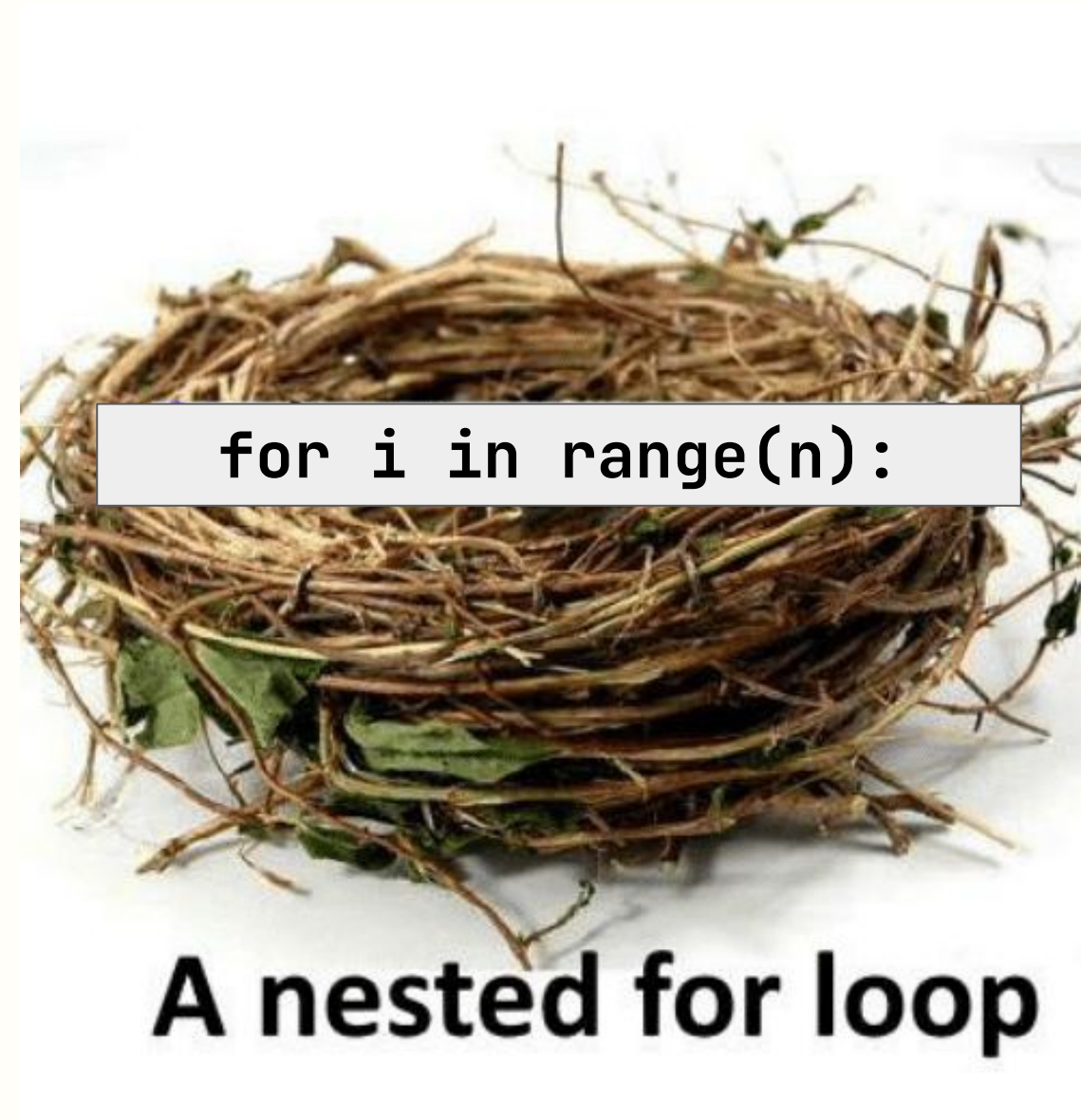
```
filtered_squares = []
for x in range(20):
    if x % 3 == 0:
        filtered_squares.append(x * x)
```





NESTED LOOPS

- Loops within loops ... within loops ... within loops ...
- Let's see an example of what this looks like (and why we want it)





EXAMPLE: NESTED LOOP

```
for i in range(5):  
    line = ''  
    for j in range(i):  
        line += '*'  
    print(line)
```





EXAMPLE: NESTED LOOP

```
for i in range(5):  
    line = ''  
    for j in range(i):  
        line += '*'  
    print(line)
```

Output:

—





EXAMPLE: NESTED LOOP

```
for i in range(5):  
    line = ''  
    for j in range(i):  
        line += '*'  
    print(line)
```

Output:

```
—  
*  
*  
*  
*  
*
```





EXAMPLE: NESTED LOOP

```
for i in range(5):  
    line = ''  
    for j in range(i):  
        line += '*'  
    print(line)
```

Output:

```
—  
*  
**
```





EXAMPLE: NESTED LOOP

```
for i in range(5):  
    line = ''  
    for j in range(i):  
        line += '*'  
    print(line)
```

Output:

```
—  
*  
**  
***
```





EXAMPLE: NESTED LOOP

```
for i in range(5):  
    line = ''  
    for j in range(i):  
        line += '*'  
    print(line)
```

Output:

```
—  
*  
**  
***  
****
```





THE WHILE LOOP

So we've seen how to iterate over a sequence, but what if there's not a particular sequence we have in mind? For example, what if we want to keep doing something until the user clicks on the screen?



The while loop repeats **while** a condition is true (i.e., until it's false)

```
while [condition]:  
    # Do something
```

We used a while loop previously for animation. Can you explain how it works now?





ITERATION PRACTICE

Write a script that prints out the number of digits in a string `s`.

For example, for `"a9Bz80d"` you should output 3.

Hint: What kind of problem is taking a whole sequence and reducing it to one value?

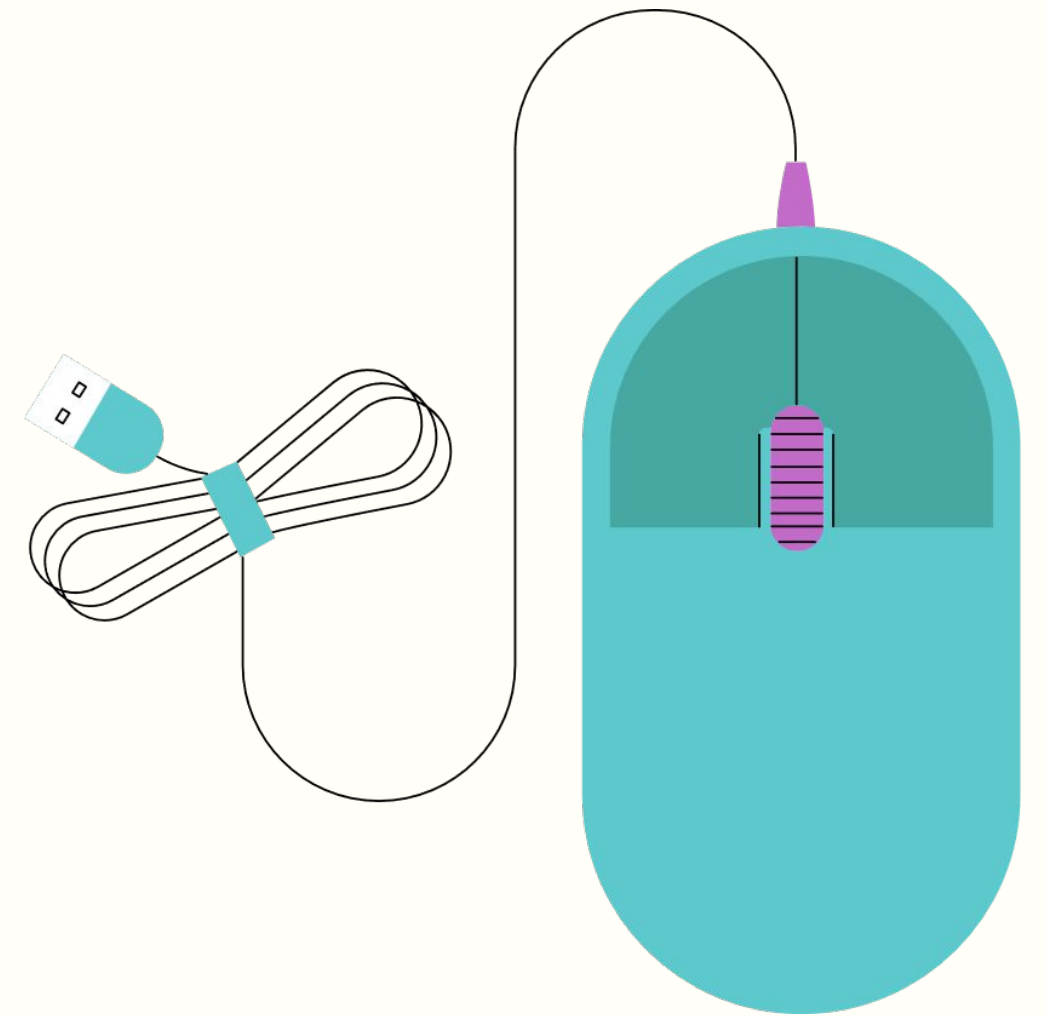




04

WORKSHEET

Let's try a harder problem!





“That’s all Folks!”



CREDITS



This presentation template is free for everyone to use thanks to the following:

SlidesCarnival for the presentation template

Pexels for the photos

HAPPY DESIGNING!