

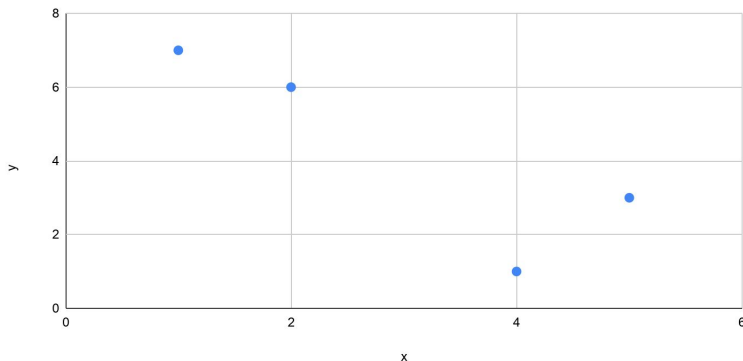
# Data Structures

# Data Structures

What kinds of data have we learned about so far?

What do you think the term **data structure** means?

Given what we know now, how would we store the coordinates of a bunch of points on a graph?



# Data Structures

So far, we've learned about several basic data types - `int`, `float`, `string`, and `bool`.

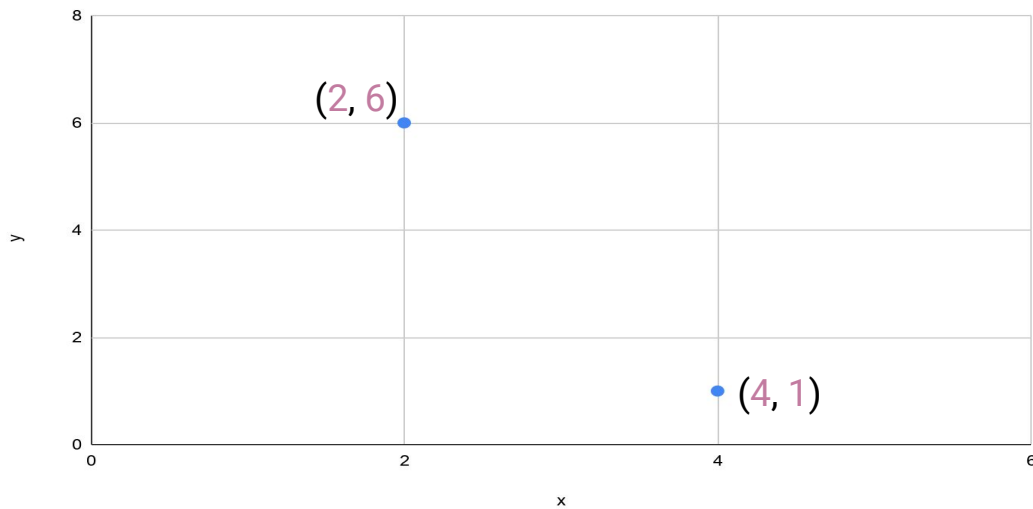
A lot of information in the world is way more complicated than that though - things like points on a graph, phone numbers tied to a name, or even a game like Tic Tac Toe.

A lot of that information can't be stored in the basic data types we've learned so far.

Let's discuss on the next slide some different ways we could store information like points on a graph.

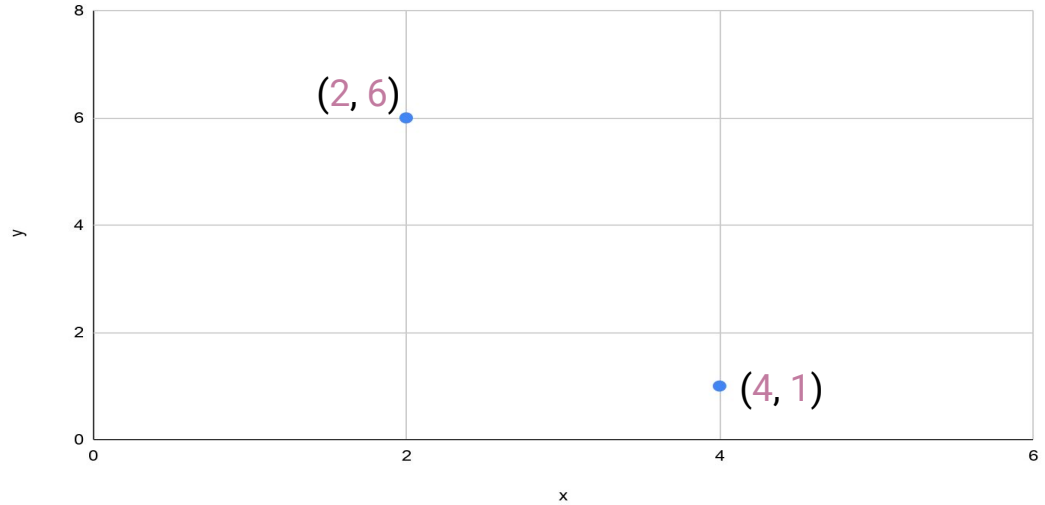


# Storing Graph Points



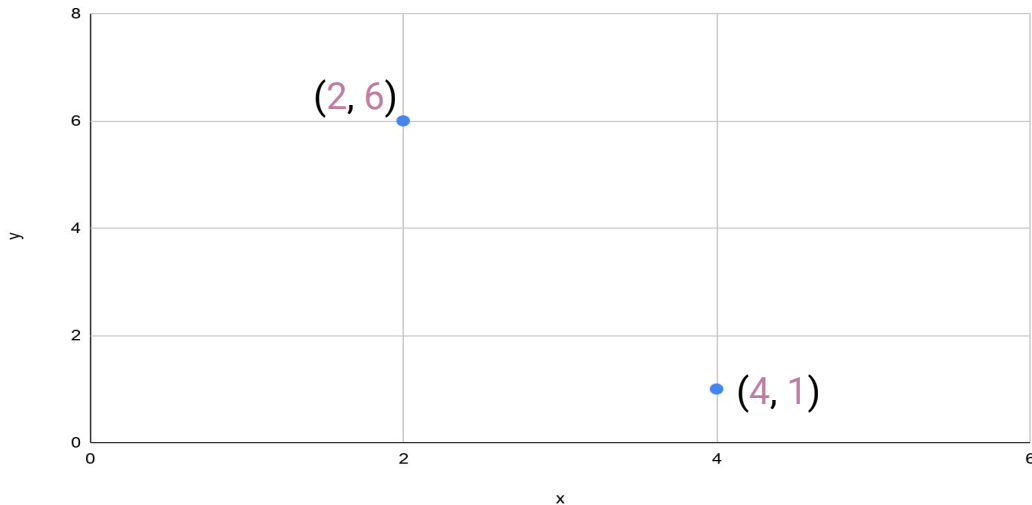
# Storing Graph Points

```
x_coor1 = 4  
y_coor1 = 1  
x_coor2 = 2  
y_coor2 = 6
```



# Storing Graph Points

```
x_coord1 = 4  
y_coord1 = 1  
x_coord2 = 2  
y_coord2 = 6  
  
coord1 = "4, 1"  
coord2 = "2, 6"  
distance = ???
```



$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

# How to actually store it!

Today we're learning about a data structure called a **tuple**.

According to CodeHS, a **tuple** is "a **heterogenous, immutable data type that stores an ordered sequence of things**".

I'll go into more detail about that whole thing soon, but here's how it looks when we make one:

```
coord1 = (4, 1)
```

```
coord2 = (2, 6)
```




# "But don't parentheses mean functions?"

I hear you ask.

**Usually**, yes! There is a big difference between calling a **function** and creating a **tuple** though - let's see if you can see it.

```
one = point(4, 2)
two = (4, 2)
```

The difference is that when we make a **tuple**, we aren't saying a name - we **just** have parentheses.

A decorative graphic in the bottom right corner consisting of several overlapping green triangles and rectangles in various shades of green.



# An Ordered Sequence of Things

A **tuple** can store a whole bunch of values, and it will keep them in the order they were put in.

```
my_tup = (5, 8, 2, 59, 9, -3)
```

Each thing inside of a **tuple** is called an **element**. Every **element** in a **tuple** has a unique number associated with it, called its **index**. An **element's index** tells us where we can find that **element** within the **tuple**.



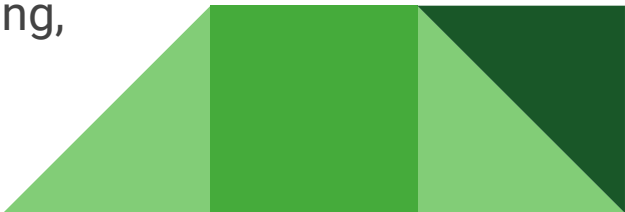
# An Ordered Sequence of Things

A **tuple** can store a whole bunch of values, and it will keep them in the order they were put in.

	0	1	2	3	4	5
<code>my_tup</code>	5	8	2	59	9	-3

Each thing inside of a **tuple** is called an **element**. Every **element** in a **tuple** has a unique number associated with it, called its **index**. An **element's index** tells us where we can find that **element** within the **tuple**.

Counting for indices, like many other things in programming, starts at **zero**.

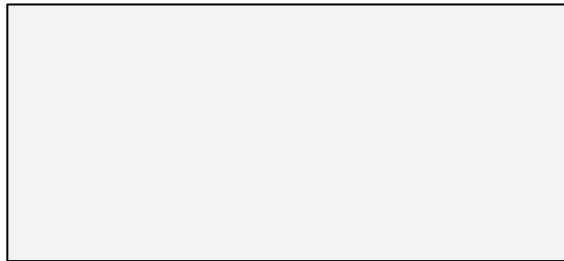


# An Ordered Sequence of Things

	Indices for <code>my_tup</code>										
	0	1	2	3	4	5					
<code>my_tup</code> =	(5	,	8	,	2	,	59	,	9	,	-3)

If we want to find the value at any given index within a **tuple**, we can do so by using square brackets `[]` with the desired index inside after the name of the **tuple**, like so:

```
print(my_tup[3])
```



# An Ordered Sequence of Things

	Indices for <code>my_tup</code>											
	0	1	2	3	4	5						
<code>my_tup</code> =	(5	,	8	,	2	,	59	,	9	,	-3	)

If we want to find the value at any given index within a **tuple**, we can do so by using square brackets `[]` with the desired index inside after the name of the **tuple**, like so:

```
print(my_tup[3])
```



59



# An Ordered Sequence of Things

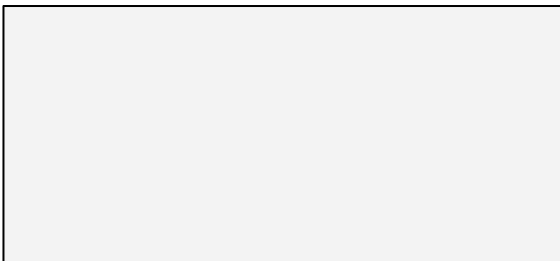
Indices for `my_tup`

0	1	2	3	4	5
5	8	2	59	9	-3

`my_tup = (5, 8, 2, 59, 9, -3)`

If we want to grab more than 1 **element** from the **tuple** at a time, we can take a **slice**. This is done by placing 2 indices, separated by a `:`, in the square brackets after the **tuple**'s name. Like the `range()` function, the second index **won't** be included! A **slice** of a **tuple** will result in another **tuple**.

```
print(my_tup[1:5])
```



# An Ordered Sequence of Things

	Indices for <code>my_tup</code>										
	0	1	2	3	4	5					
<code>my_tup</code> =	(5	,	8	,	2	,	59	,	9	,	-3)

If we want to grab more than 1 **element** from the **tuple** at a time, we can take a **slice**. This is done by placing 2 indices, separated by a `:`, in the square brackets after the **tuple**'s name. Like the `range()` function, the second index **won't** be included! A **slice** of a **tuple** will result in another **tuple**.

```
print(my_tup[1:5])
```

```
(8, 2, 59, 4)
```

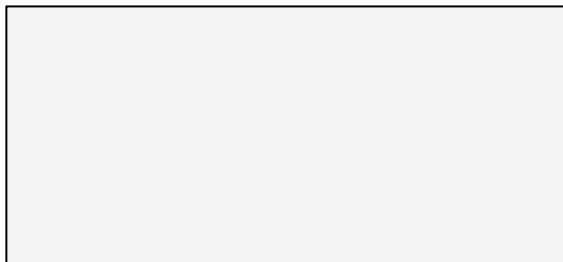


# An Ordered Sequence of Things

	Indices for <code>my_tup</code>										
	0	1	2	3	4	5					
<code>my_tup</code> =	(5	,	8	,	2	,	59	,	9	,	-3)

If we want to find the number of **elements** in a **tuple**, we can use the `len()` function, which is built into Python!

```
print(len(my_tup))
```



# An Ordered Sequence of Things

	Indices for <code>my_tup</code>										
	0	1	2	3	4	5					
<code>my_tup</code> =	(5	,	8	,	2	,	59	,	9	,	-3)

If we want to find the number of **elements** in a **tuple**, we can use the `len()` function, which is built into Python!

```
print(len(my_tup))
```

6

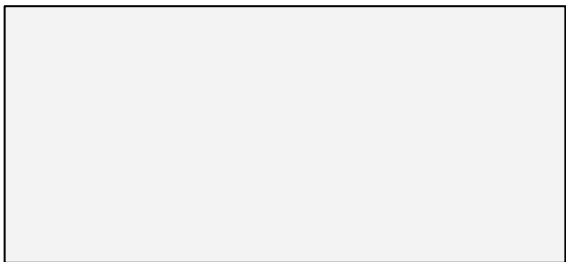




# Adding Tuples

**Tuples** can be concatenated together!

```
tup = (1, 2)
twop = (3, 4)
tup3 = tup + twop
print(tup3)
```

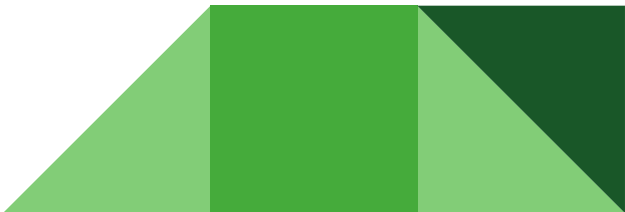


# Adding Tuples

**Tuples** can be concatenated together!

```
tup = (1, 2)
twop = (3, 4)
tup3 = tup + twop
print(tup3)
```

```
(1, 2, 3, 4)
```

A decorative graphic in the bottom right corner consisting of several green geometric shapes: a light green triangle, a medium green square, and a dark green triangle.

# Adding Tuples

**Tuples** can be concatenated together!

```
tup = (1, 2)
twop = (3, 4)
tup3 = tup + twop
print(tup3)
```

```
(1, 2, 3, 4)
```

Like **strings**, however, you cannot concatenate **tuples** with other data types.

```
tup = (1, 2)
print(tup + 3)
```



# More 'bout Tuples

One feature of **tuples** is that they are **immutable** - you aren't allowed to change any of the individual elements by themselves. You can completely overwrite a **tuple**, but you cannot edit any one piece of data inside of it.



# Back to the definition

"**Tuples** are **heterogenous** ... "

This means that the types of the elements in a **tuple** **do not** have to all be the same type. For example:

```
tup = (1, "two", 3.0, False)
```

This is a valid **tuple**!

You can even use **tuples** as elements in other **tuples**!

```
tup = (1, "two", 3.0, False, (5, "six"))
```

```
print(tup[4][0])      # 5
```



# Tuples with few elements

We can create an empty **tuple** by just putting nothing inside the parentheses - like so:

```
t_empty = ()
```

We can't access anything from inside an empty **tuple** - there's nothing inside them to access!

We can also make **tuples** with only 1 element - but not exactly how you'd expect!



# Tuples with 1 element

```
t_one = (5)
```

Because Python can use parentheses in 3 ways (**functions**, **tuples**, and **math**), it will always default to **math** first. So unless we include a symbol that is never used in math, Python will assume we're trying to do math.

In order to make a **tuple** with 1 element, we need to include a comma, like so:

```
t_one = (5,)
```

