```
{

    \\Lists in Python;

}
```

Lesson‘Objective’;

In this lesson you’ll learn how to use Python lists to store ordered collections of values.

Lists are incredibly useful when writing code to manage several related variables.

01 {

[List]

< Lists in Python represent ordered sequences of values. >

}

Concepts < /1 > {

```
primes = [2, 3, 5, 7]
```

}

Concepts < /2 > {

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
```

}

Concepts < /3 > {

```
hands = [
    ['J', 'Q', 'K'],
    ['2', '2', '2'],
    ['6', 'A', 'K'], # (Comma after the last element is optional)
]
# (I could also have written this on one line, but it can get hard to read)
hands = [['J', 'Q', 'K'], ['2', '2', '2'], ['6', 'A', 'K']]
```

}

Concepts < /4 >{

```
my_favourite_things = [32, 'raindrops on roses', help]
# (Yes, Python's help function is *definitely* one of my
favourite things)
```

}

02 {

[Indexing]

< You can access individual list elements with **square brackets**.>

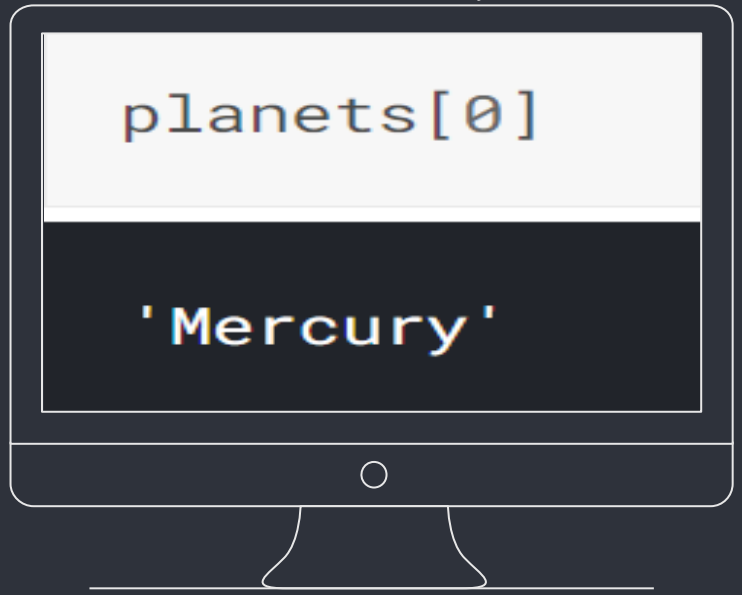
}

Indexing{

Which planet is closest to the sun? Python uses zero-based indexing, so the first element has index 0.

}

```
planets = ['Mercury', 'Venus',
'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
```

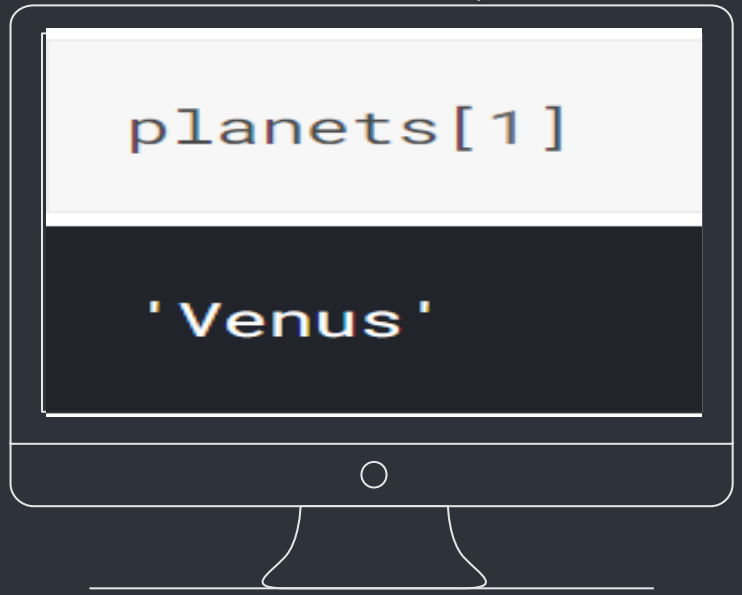


Indexing{

What's the next closest planet?

}

```
planets = ['Mercury', 'Venus',
'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
```



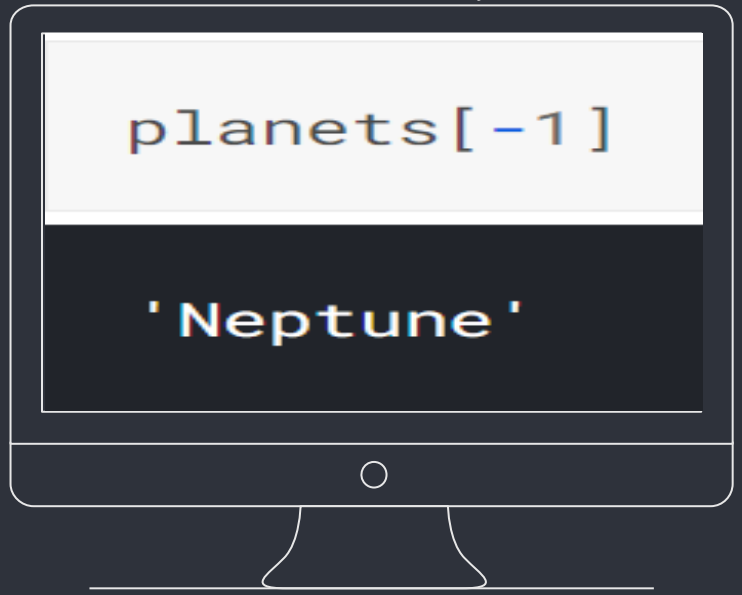
Indexing{

Which planet is *furthest* from the sun?

Elements at the end of the list can be accessed with negative numbers, starting from -1:

}

```
planets = ['Mercury', 'Venus',
'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
```



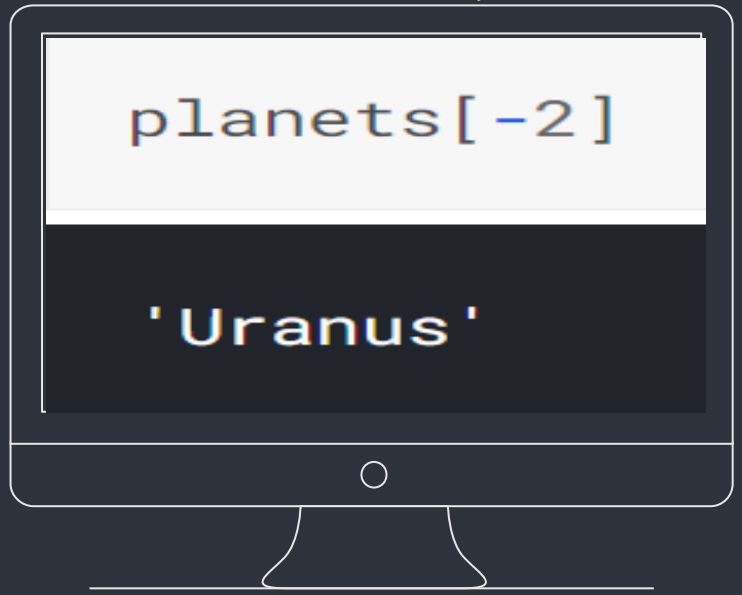
Indexing{

Which planet is *furthest* from the sun?

Elements at the end of the list can be accessed with negative numbers, starting from -1:

}

```
planets = ['Mercury', 'Venus',
'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
```



03{

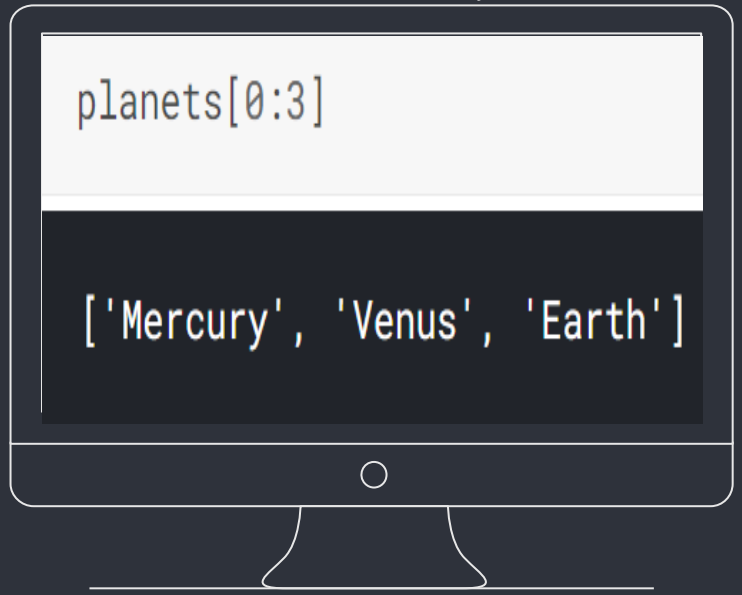
[Slicing]

< What are the first three planets? *We can answer this question using* ***slicing***>

}

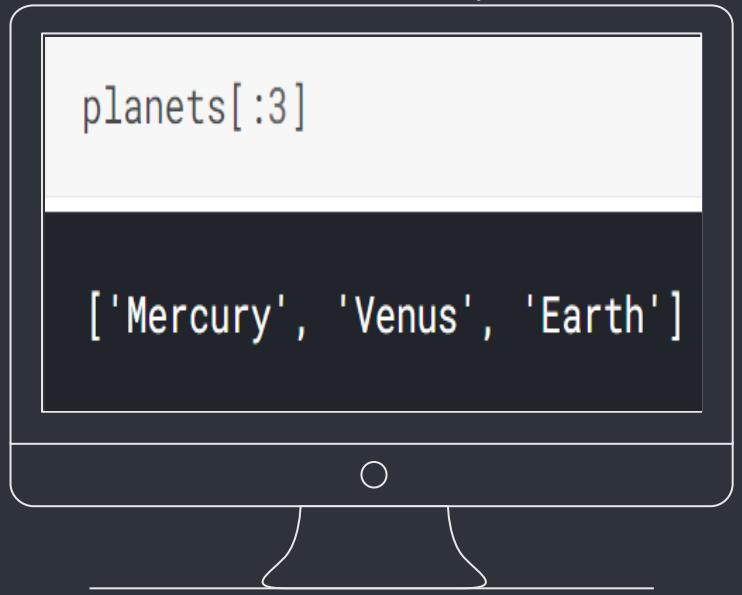
```
1
2
3
4  Slicing{
5
6
7
8
9      What are the first
10     three planets?
11
12
13
14 }
```

```
planets = ['Mercury', 'Venus',
'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
```



```
1
2
3
4  Slicing{
5
6
7
8
9      The starting and
10     ending indices are
11     both optional.
12
13
14 }
```

```
planets = ['Mercury', 'Venus',
'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
```

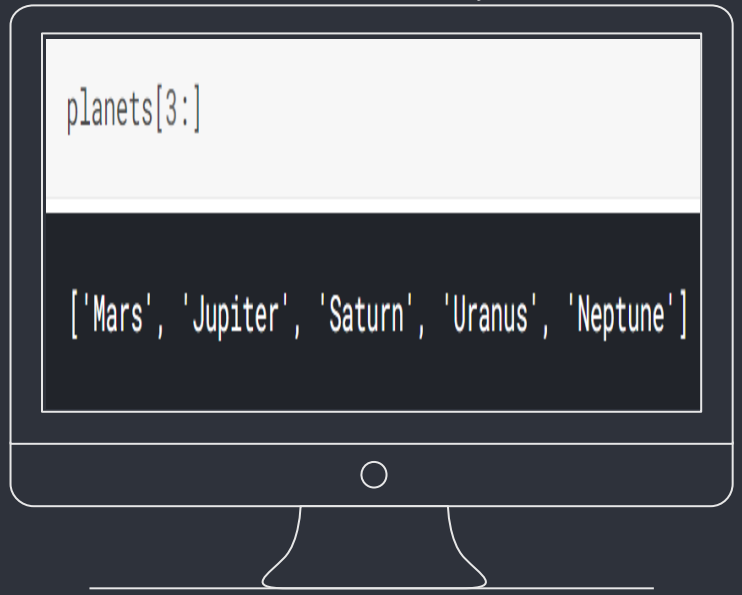


Slicing{

If we leave out the end index, it's assumed to be the length of the list.

}

```
planets = ['Mercury', 'Venus',
'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
```

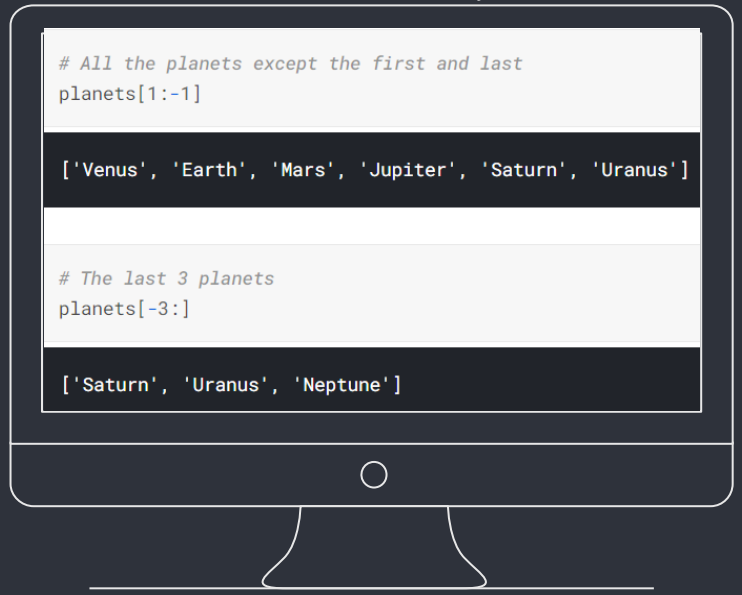


Slicing{

We can also use negative indices when slicing.

}

```
planets = ['Mercury', 'Venus',
'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
```



04{

[Changing lists]

< Lists are "mutable", meaning they can be modified "in place".>

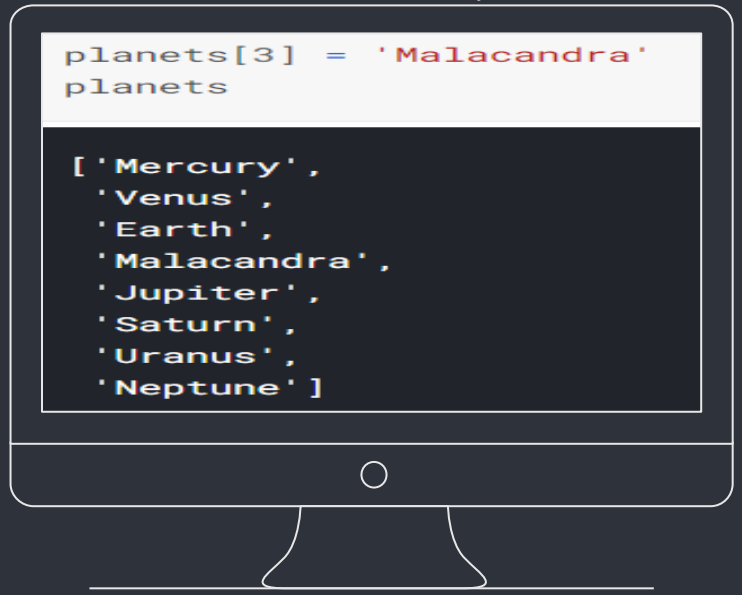
}

Changing lists

One way to modify a list is to assign to an index or slice expression.

For example, let's say we want to rename Mars:

```
planets = ['Mercury', 'Venus',
'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
```



Changing lists

One way to modify a list is to assign to an index or slice expression.

Let's compensate by shortening the names of the first 3 planets.

```
planets = ['Mercury', 'Venus',
'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
```

05{

[List functions]

< Python has several useful functions for working with lists.>

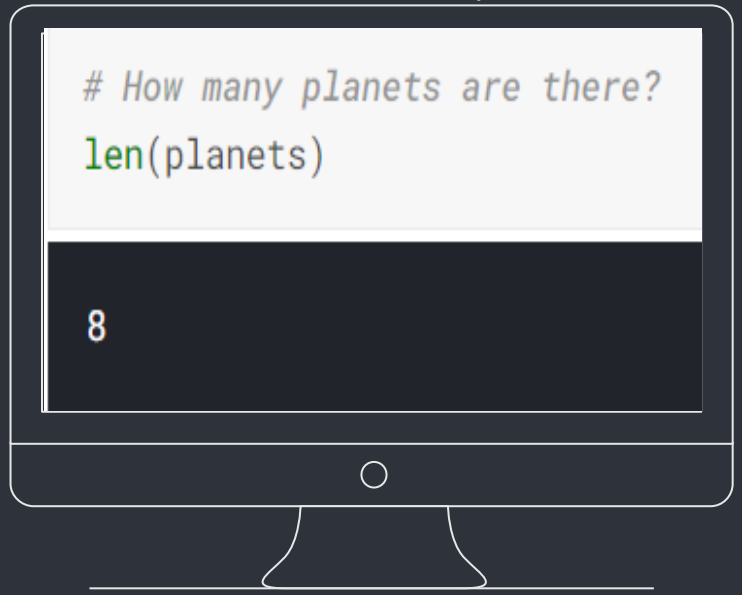
}

List functions{

len gives the length of a list:

}

```
planets = ['Mercury', 'Venus',
'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
```



List functions{

sorted gives a sorted version of a list:

}

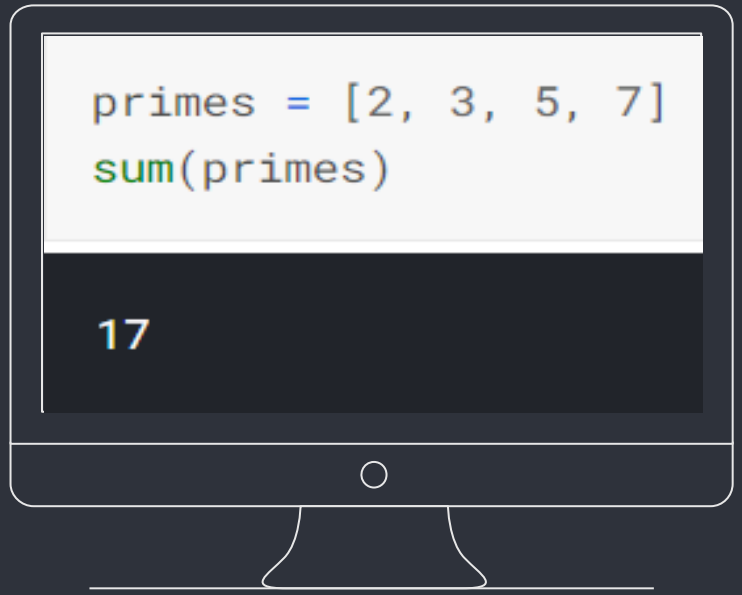
```
planets = ['Mercury', 'Venus',
'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
```

List functions{

sum does what you might expect:

}

```
primes = [2, 3, 5, 7]
```

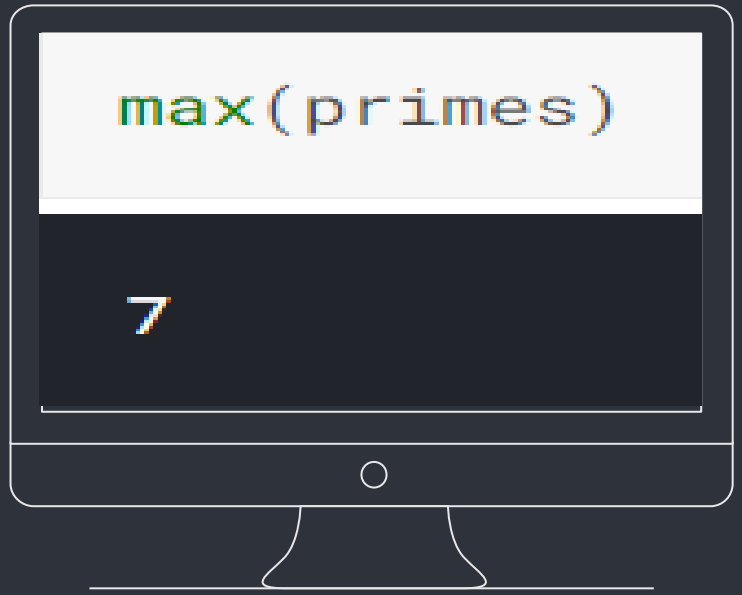


List functions{

We've previously used the **min** and **max** to get the minimum or maximum of several arguments. But we can also pass in a single list argument.

}

```
primes = [2, 3, 5, 7]
```

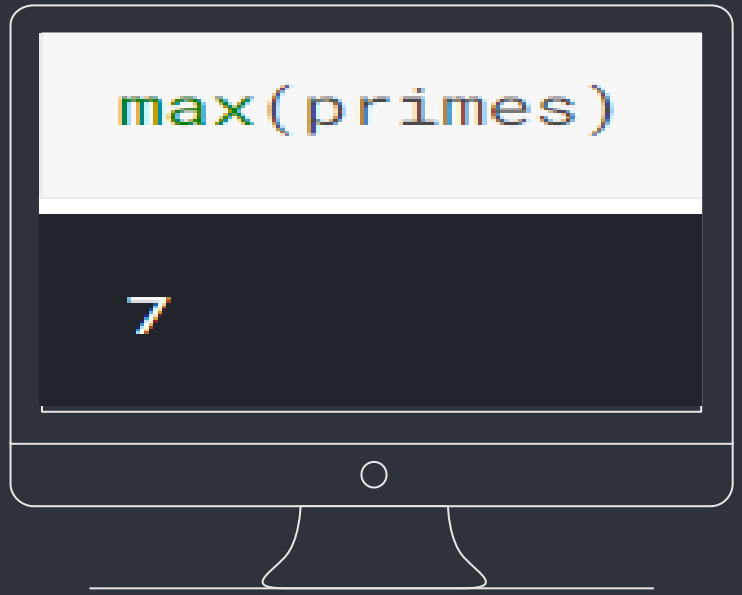


List functions{

We've previously used the **min** and **max** to get the minimum or maximum of several arguments. But we can also pass in a single list argument.

}

```
primes = [2, 3, 5, 7]
```



06{

[Interlude: Objects]

< We've used the term 'object' a lot so far - you may have even read that *everything* in Python is an object. What does that mean?

In short, objects carry some things around with them. You access that stuff using Python's **dot** syntax.>

}

Objects{

For example, numbers in Python carry around an associated variable called **imag** representing their **imaginary part.**

(You'll probably never need to use this unless you're doing some very weird math.)

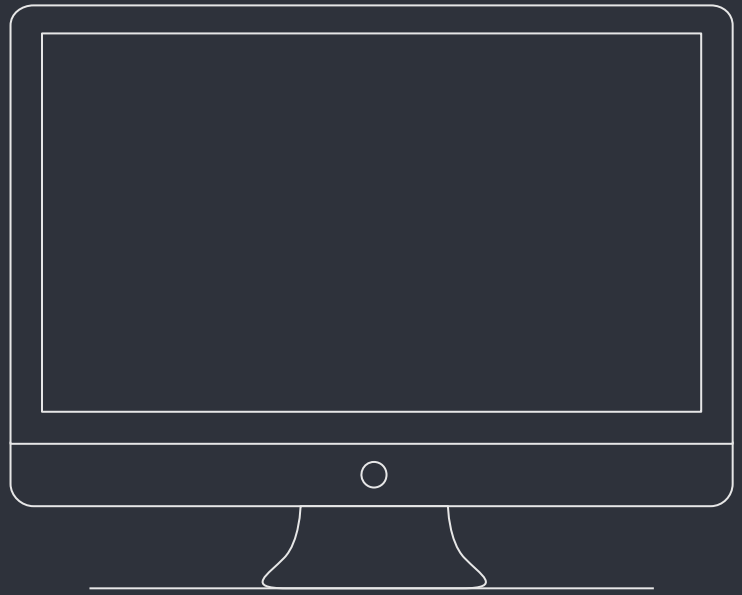
}

Objects

The things an object carries around can also include *functions*.

A *function* attached to an object is called a **method**.

(Non-function things attached to an object, such as *imag*, are called *attributes*).

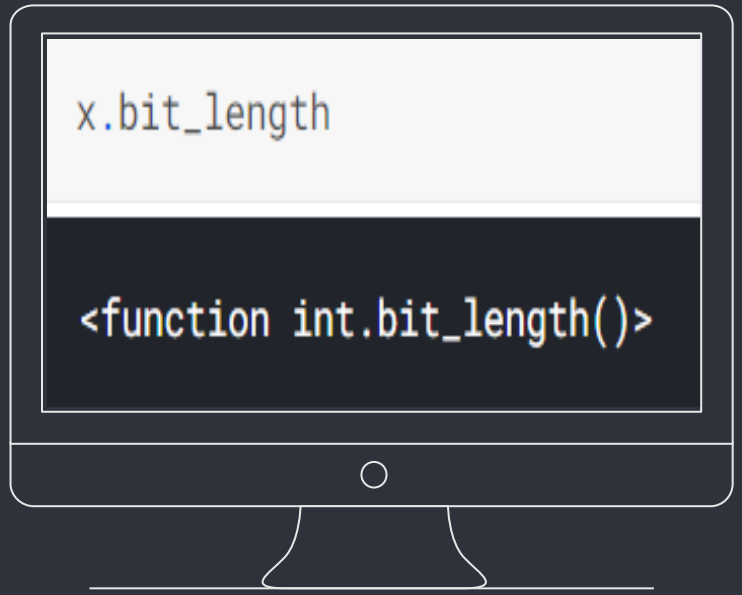


Objects{

For example, numbers have a method called **bit_length**.

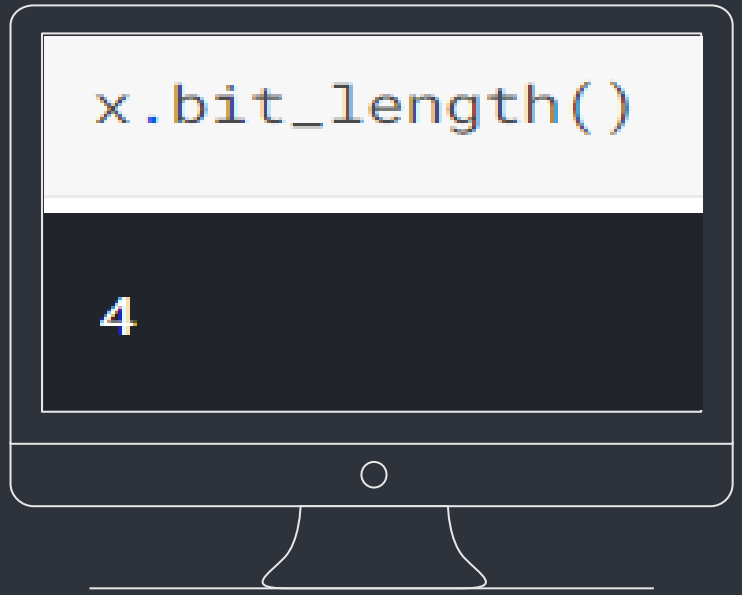
Again, we access it using dot syntax:

}



Objects{

To actually call it, we add parentheses:

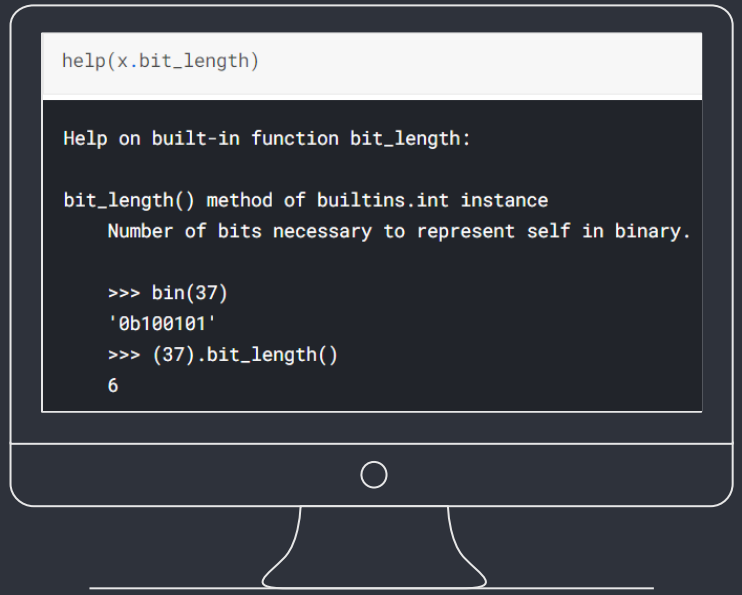


}

Objects{

In the same way that we can pass functions to the help function (e.g. help(max)), we can also pass in methods:

}



07{

[List methods]

< The examples in the previous slides were utterly obscure. None of the types of objects we've looked at so far (numbers, functions, booleans) have attributes or methods you're likely ever to use.

But it turns out that lists have several methods which you'll use all the time.>

}

List methods{

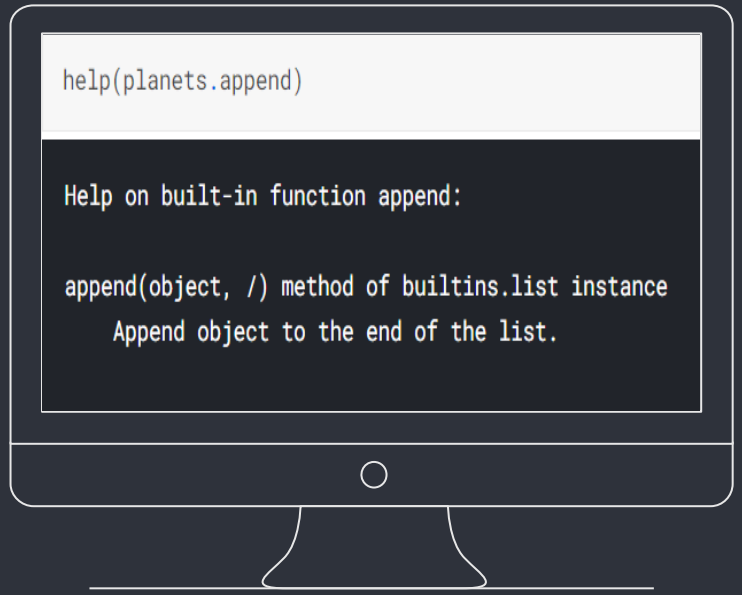
list.append modifies a list by adding an item to the end:

}

List methods{

Why does the cell above have no output? Let's check the documentation by calling **help(planets.append).**

}



List methods{

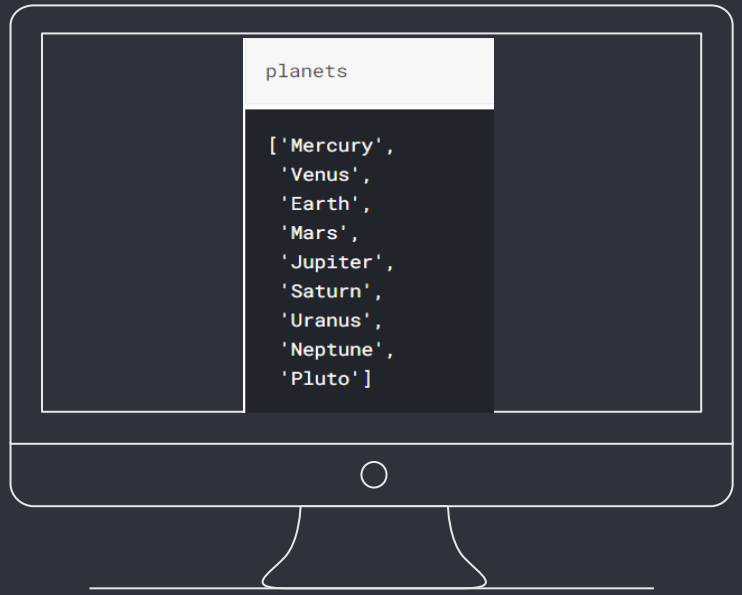
append is a method carried around by *all* objects of type list, not just **planets**, so we also could have called **help(list.append)**. However, if we try to call **help(append)**, Python will complain that no variable exists called "append". The "append" name only exists within lists - it doesn't exist as a standalone name like builtin functions such as **max** or **len**.

}

List methods{

The **-> None** part is telling us that **list.append** doesn't return anything. But if we check the value of **planets**, we can see that the method call modified the value of **planets**:

}



List methods{

list.pop removes and returns the last element of a list:

}

```
planets.pop()
```

```
'Pluto'
```

```
planets
```

```
['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

08{

[Searching list]

< Where does Earth fall in the order of planets? We can get its index using the **list.index** method.>

}

Searching list{

Where does Earth fall in the order of planets? We can get its index using the **list.index** method.

}

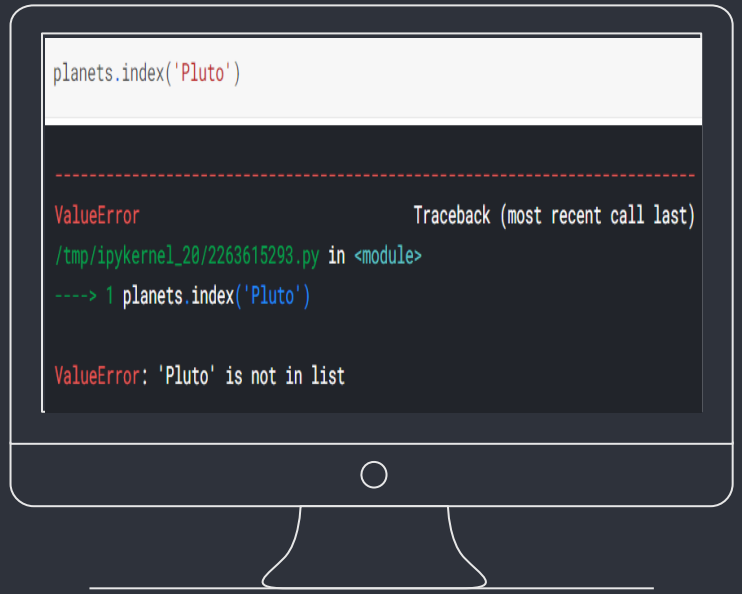
```
planets.index('Earth')
```

```
2
```

Searching list{

At what index does Pluto occur?

}



Searching list{

To avoid unpleasant surprises like this, we can use the in operator to determine whether a list contains a particular value:

}

```
# Is Earth a planet?
"Earth" in planets
```

```
True
```

```
# Is Calbefraques a planet?
"Calbefraques" in planets
```

```
False
```

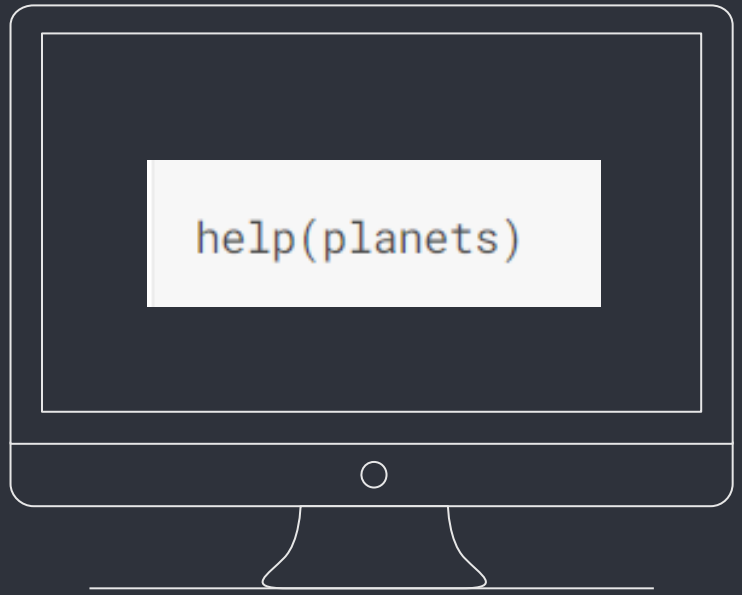
Searching list{

There are a few more interesting list methods we haven't covered.

If you want to learn about all the methods and attributes attached to a particular object, we can call help() on the object itself.

For example, **help(planets)** will tell us about all the list methods:

}



09{

[Tuples]

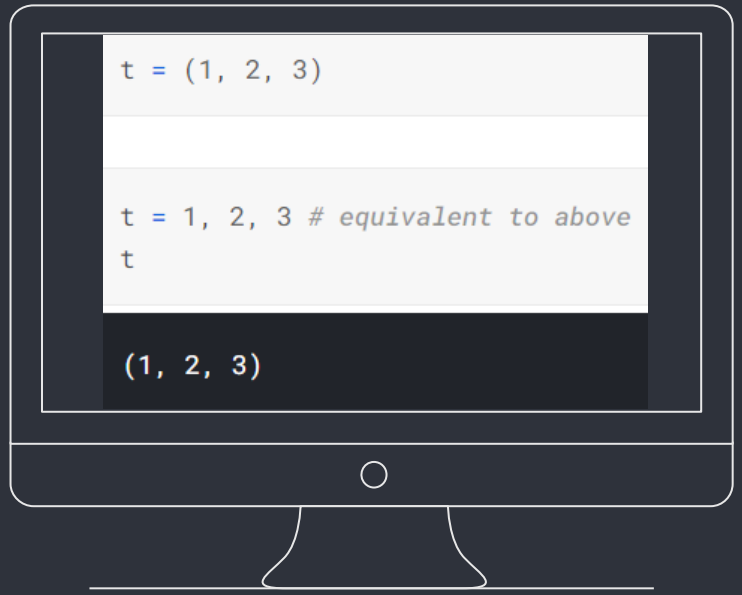
< Tuples are almost exactly the same as lists. They differ in just two ways:
1: The syntax for creating them uses parentheses instead of square brackets
2: They cannot be modified (they are immutable).>

}

Tuples{

1: The syntax for creating them uses parentheses instead of square brackets

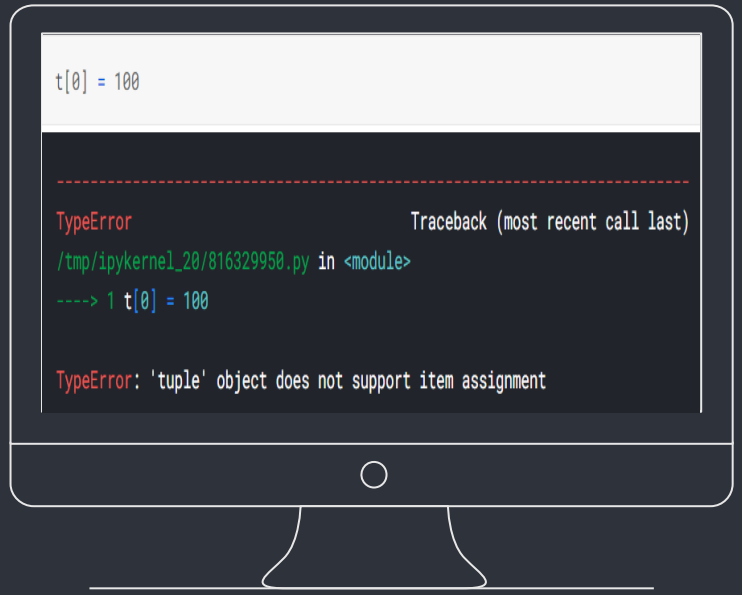
}



Tuples{

2: They cannot be modified (they are immutable).

}

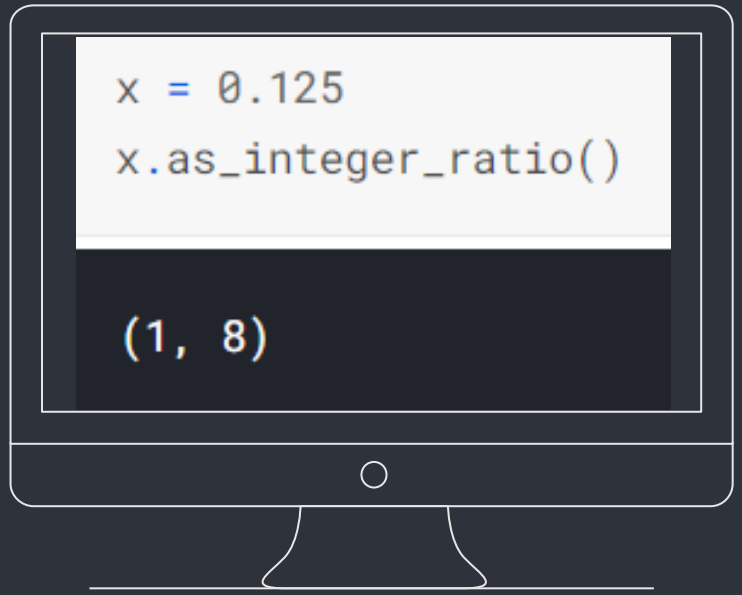


Tuples{

Tuples are often used for functions that have multiple return values.

For example, the **as_integer_ratio()** method of float objects returns a numerator and a denominator in the form of a tuple:

}



Tuples{

These multiple return values can be individually assigned as follows:

}

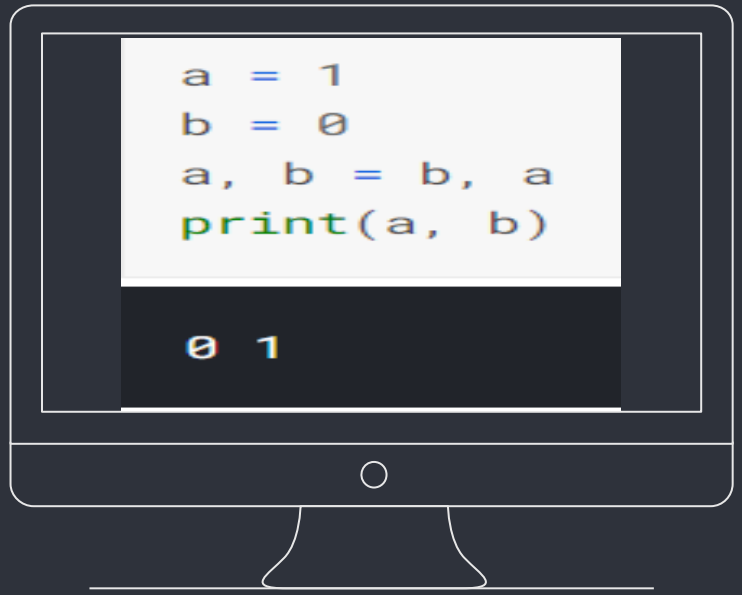
```
numerator, denominator = x.as_integer_ratio()
print(numerator / denominator)
```

```
0.125
```

Tuples{

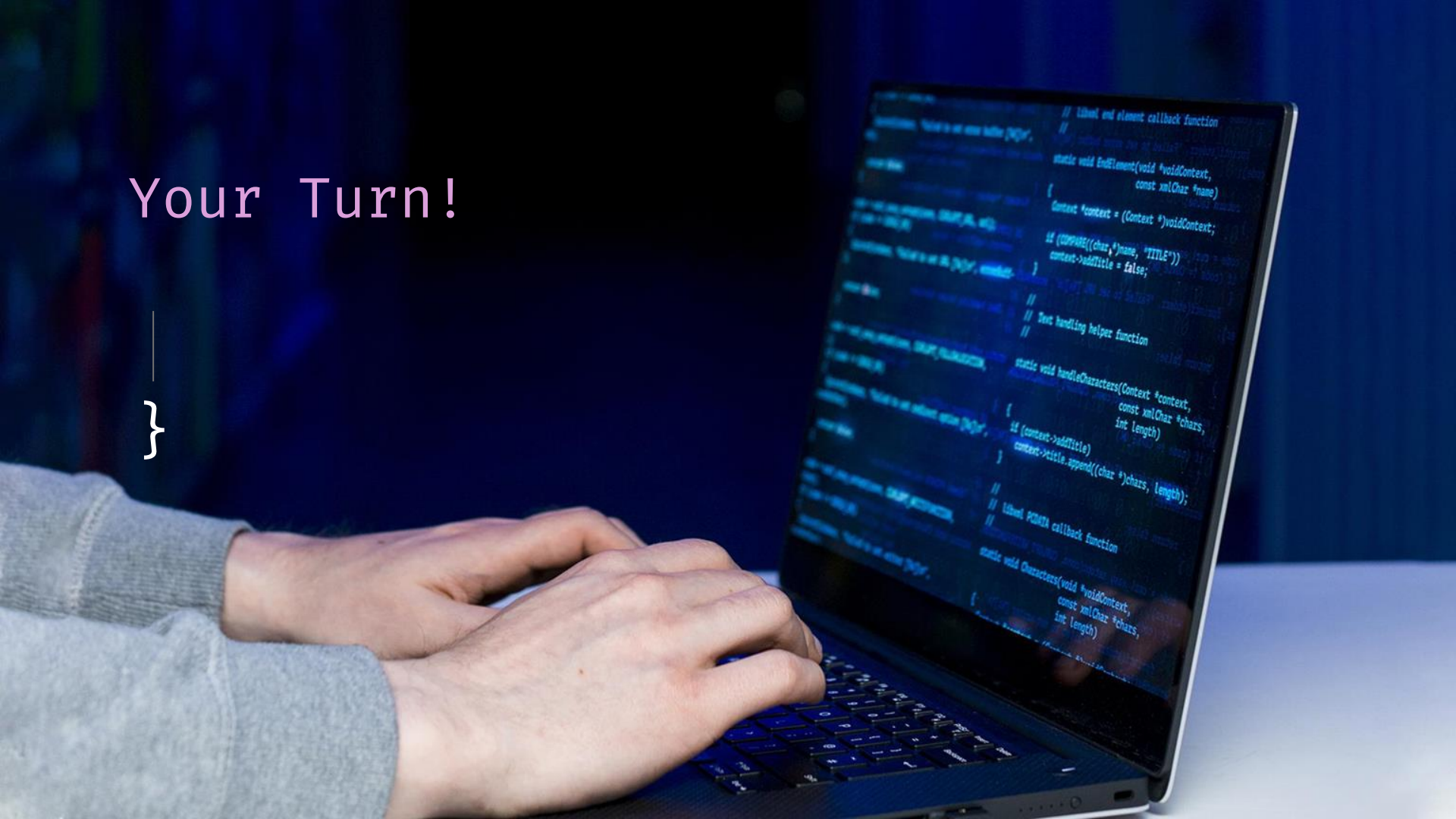
Finally we have some insight into the classic Stupid Python Trick™ for swapping two variables!

}



Your Turn!

}



01. {

Complete the function below according to its docstring.

}

```python
def select_second(L):
    """Return the second element of the given list. If the list has no second
    element, return None.
    """
    pass
```

02. {

You are analyzing sports teams. Members of each team are stored in a list. The Coach is the first name in the list, the captain is the second name in the list, and other players are listed after that. These lists are stored in another list, which starts with the best team and proceeds through the list to the worst team last. Complete the function below to select the captain of the worst team.

}

```python
def losing_team_captain(teams):
    """Given a list of teams, where each team is a list of names, return the 2nd player (captain)
    from the last listed team
    """
    pass
```

03. {

The next iteration of Mario Kart will feature an extra-infuriating new item, the Purple Shell. When used, it warps the last place racer into first place and the first place racer into last place. Complete the function below to implement the Purple Shell's effect.

}

```python
def purple_shell(racers):
    """Given a list of racers, set the first place racer (at the front of the list) to last
    place and vice versa.

    >>> r = ["Mario", "Bowser", "Luigi"]
    >>> purple_shell(r)
    >>> r
    ["Luigi", "Bowser", "Mario"]
    """
    pass
```

04. {

What are the lengths of the following lists? Fill in the variable **lengths** with your predictions. (Try to make a prediction for each list *without* just calling **len()** on it.)

}

```
a = [1, 2, 3]
b = [1, [2, 3]]
c = []
d = [1, 2, 3][1:]

# Put your predictions in the list below. Lengths should contain 4 numbers, the
# first being the length of a, the second being the length of b and so on.
lengths = []
```

05. {

We're using lists to record people who attended our party and what order they arrived in. For example, the following list represents a party with 7 guests, in which Adela showed up first and Ford was the last to arrive:

party_attendees = ['Adela', 'Fleda', 'Owen', 'May', 'Mona', 'Gilbert', 'Ford’]

A guest is considered 'fashionably late' if they arrived after at least half of the party's guests. However, they must not be the very last guest (that's taking it too far). In the above example, Mona and Gilbert are the only guests who were fashionably late.

Complete the function below which takes a list of party attendees as well as a person, and tells us whether that person is fashionably late.

}

```
def fashionably_late(arrivals, name):
    """Given an ordered list of arrivals to the party and a name, return whether the guest with that
    name was fashionably late.
    """
    pass
```

Thanks; {

‘Do you have any questions?’

tahil.salimar@wmsu.edu.ph
salimar.tahil@wmsu.edu.ph -> this is where you’ll send your activities.

}