# Python Fundamentals

Lists, for statement, range function, break, continue

# Objectives:

- What is a list?
- Using the for statement to traverse a list
- range function
- break and continue

# Lists

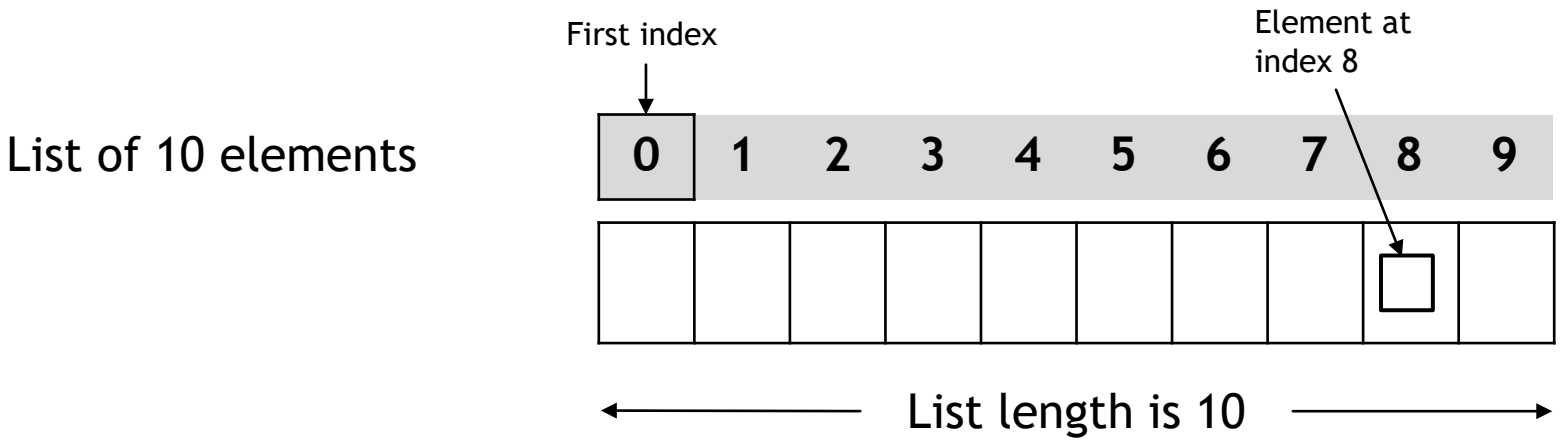
It will often be the case when writing our code that we'll want to store a collection of values.  All the names in a class, a list of movies, and so on.

Now while we could create a variable to store each and every name or movie, this would quickly get very long-winded, and become difficult to manage.

This is where lists come in.

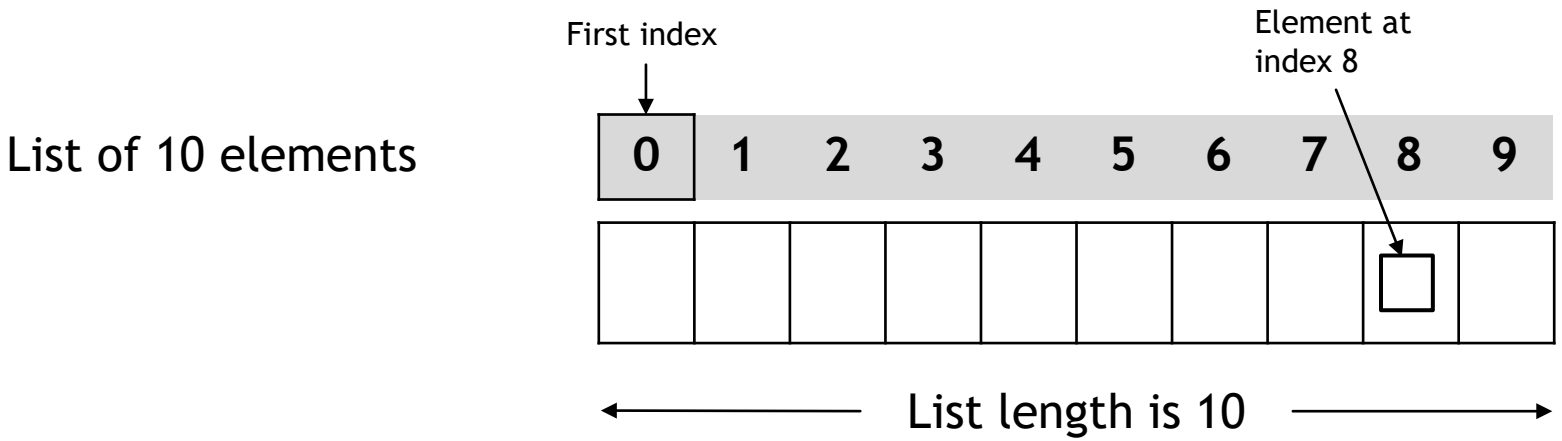
# What is a list?

A list is a container object that holds a varying number of values.  A list has an initial size, but elements can be added or removed over time.



# What is a list?

We call each item in a list an **element**. Each element is accessed by a numerical **index**. As shown below, index positions begin at 0. The 9th *element*, for example, has an *index* of 8.



# Creating a list

A list variable is declared like any other variable. The only difference is the way we declare the initial value. Below are some examples of declaring lists.

```
# An empty list
empty_list = []

# A list of integers
number_list = [1, 2, 3, 4, 5]

# A list of strings
string_list = ["Orange", "Red", "Blue", "Green"]
```

# Creating a list

Python has no rules around the type of values a list can store.  It could be a numeric type, a string, a Boolean, or something else.

```
# A list of mixed types
mixed_list = [1, 2.0, "Hello", True, 100, -5]

# Another list of mixed types
another_mixed_list = ["Blah", "Bloo", False, 4.5, 3j]

print(mixed_list)
# [1, 2.0, 'Hello', True, 100, -5]
```

# Accessing elements in a list

As mentioned, every element has an index position. We use that index position to retrieve a value from a list, or assign a value to an existing position. Below are some examples.

```
even_numbers = [2, 4, 6, 8, 10, 12, 14, 16, 17, 20]

print("The number at index position 8 is:", even_numbers[8])
# Writes to console: "The number at index position 8 is: 17"

even_numbers[8] = 18
# Changes the element at index position 8 to the number 18
```

# Accessing elements in a list

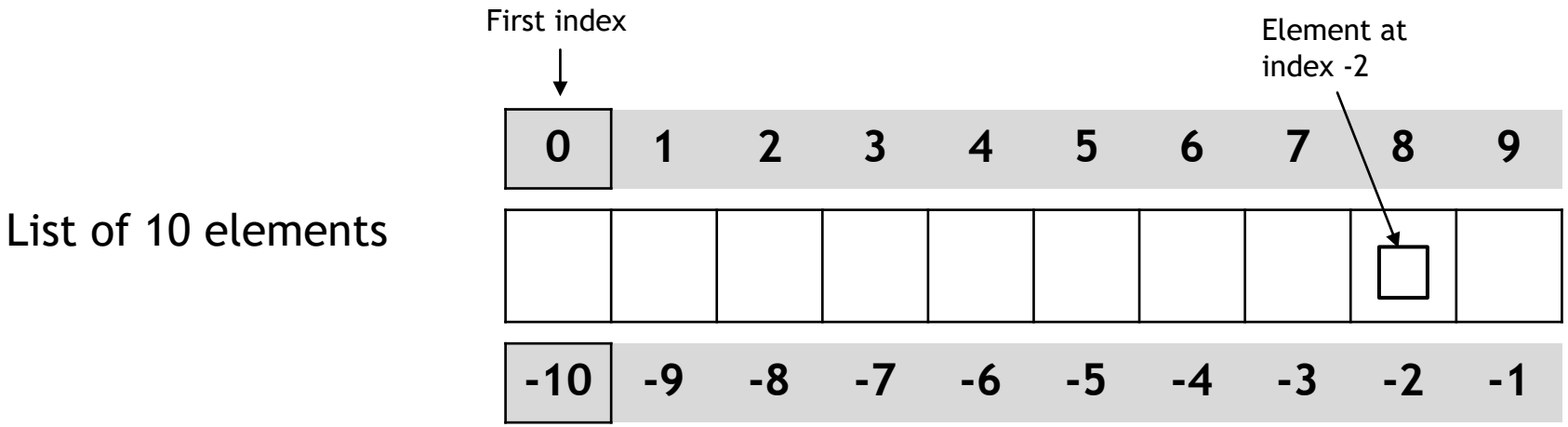
- What happens if we try to access a non existing position of the list?
- i.e. `num_list = [1, 2, 3, 4, 5, 6, 7]`
- It is clear that Python will create a list with seven elements.
- We try to assign the values 77.5 and 33.3 in the following indexes.

```
num_list[7] = 77.5
num_list[-3] = 33.3
```

→ The *first* statement will cause a run-time error. Specifically:
**IndexError: list assignment index out of range**

# Accessing elements in a list

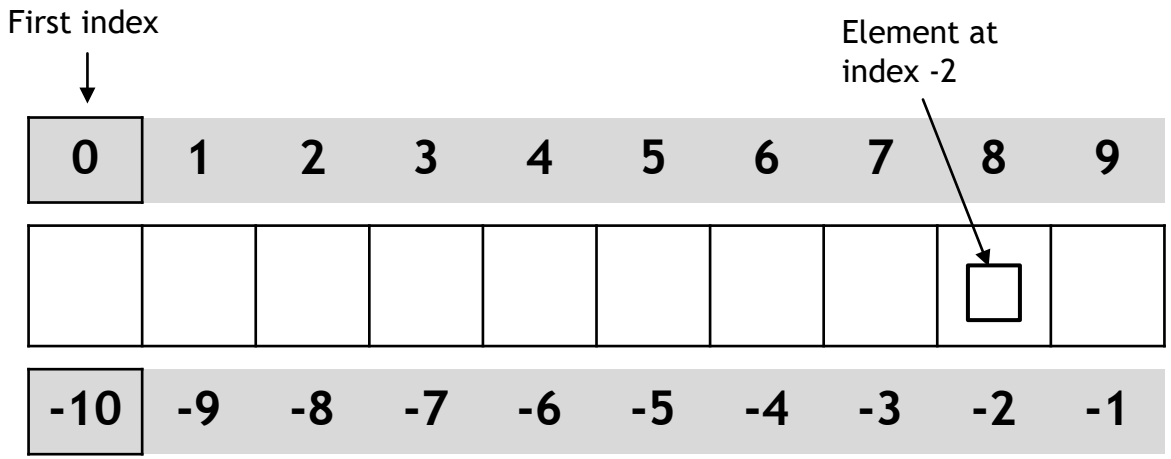
Accessing a negative index position does not, necessarily, cause an error with lists. Why is that? As well as having a positive position, every element in a list has a negative position as well. Below illustrates this.



# Accessing elements in a list

This is useful if you want to access the last element in a list, without necessarily knowing its size. Because -1 will always be the last element, -2 the second last, and so on.

List of 10 elements



# Finding the length of a list

So how do we find out the length of a list? Python provides a function called `len`, that when given a list as an argument will return its size.

```
odd_numbers = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

list_size = len(odd_numbers)

print("The list has a total of", list_size, "elements.")
# Writes to the console: "The list has a total of 10 elements."

print("The last element is:", odd_numbers[list_size - 1])
# Writes to the console: "The last element is: 19"
```

# Modifying a list

Python provides simple methods to add and remove elements from a list. We can append elements, or use pop to remove the last element.

```
cities = ["Sydney", "Melbourne", "Brisbane", "Hobart"]

cities.append("Canberra")
# List elements: ["Sydney", "Melbourne", "Brisbane", "Hobart",
    "Canberra"]

removed_city = cities.pop()
# List elements: ["Sydney", "Melbourne", "Brisbane", "Hobart"]
# The pop method also returns the removed element.
```

# del keyword

You may also want to remove an element at a particular index position. For this you can use the `del` keyword, as shown in the below example.

```
cities = ["Sydney", "Melbourne", "Brisbane", "Hobart"]

cities.append("Canberra")
# List elements: ["Sydney", "Melbourne", "Brisbane", "Hobart",
    "Canberra"]

del cities[2]
# List elements: ["Sydney", "Melbourne", "Hobart", "Canberra"]
```

# Modifying a list

A quick reminder about modifying the number of elements in a list.

This will increase or decrease the length of the list.  In the case of using `del` to remove an element, it may also adjust the index position of existing elements.

This is important to remember when accessing the list later.

# slicing a list

One final note about lists.  You are able to retrieve a section of a list using a method called *slicing*.  This will be examined more in future lessons, but below is a sample.

```
cities = ["Sydney", "Melbourne", "Brisbane", "Hobart"]

some_cities = cities[0:2]
# Retrieve from position 0 to position 2 (exclusive)
# The original list remains untouched

print(some_cities)
# Prints to the console: ['Sydney', 'Melbourne']
```

# for statement

When working with a list, there is a convenient way to iterate through each element.  This is known as the `for` statement.

```
cities = ["Sydney", "Melbourne", "Brisbane", "Hobart"]

for city in cities:
    print(city)

# Here we are looping through each element in the list and
    printing it to the console.
```

# for statement

Note that no index position is used in a `for` statement to access each element in a list.

This is because the `for` statement actually talks to what's called an **iterator**.

It's the responsibility of the **iterator** to figure out how to move through the sequence.  Keeping track of the index position and so on.

# for statement

So what does the `for` statement do?

It asks the **iterator** for the next element in a sequence. That element is assigned to a variable; in the case of our example, **city**.

It will keep asking for new elements every time it loops, until the **iterator** signals there are no elements left in the sequence.  At this point the `for` statement ends.

# for statement

Because of this, you CANNOT access the **index** position when working with a `for` statement.

This makes it useful when you want to iterate through every element in a sequence, usually for the purposes of displaying values.  In our case, printing to the console.

In all other circumstances you'll want to use a **while** loop.

# while statement with a list

Below is an example of using the while statement to move through the same list.  Note the use of a variable to keep track of the index position.

```
cities = ["Sydney", "Melbourne", "Brisbane", "Hobart"]
# Initialize a variable to store the current index
index = 0
# Obtain the size of the list
list_size = len(cities)

while index < list_size:
    print(cities[index])
    index += 1
```

# range function

While we can manually keep track of the index position with a variable, incrementing it each time we loop, Python provides an easier method.

We can make use of a `for` statement and leverage another built-in function, the `range` function.

The `range` function returns an *iterable* range of arithmetic values.  For example, an ever-increasing index position.

# for statement with the range function

Below we are leveraging `range`, using the same list of cities. And again, we use the `len` function to obtain the size of our list. Note how much more concise our code is.

```
cities = ["Sydney", "Melbourne", "Brisbane", "Hobart"]

# Use range in our for statement, along with len
# Note that the expression is only evaluated once
for i in range(len(cities)):
    print("The current index is:", i)
    print(cities[i])

# The above code would print out the values 0, 1, 2, 3
# as well as the city names
```

# range function

- You might be forgiven for thinking the range function is generating a list of numbers.  In fact, it does not.

- A range only stores **start**, **stop**, and **step** values.  Thus saving space.  It is, however, *iterable*.

- Remember iterators? In this case the range will generate a new value every time the `for` statement asks for it, until it reaches a number equal to or greater than the **stop** value.

# range function

The start, stop, and step values can be passed to the range function as arguments.  For example, our below sample prints out all odd numbers between 1 and 20.

```
# The arguments are, in order, start, stop, and step
for num in range(1, 20, 2):
    print(num)
# Prints: 1, 3, 5, 7, 9, 11, 13, 15, 17, 19

# Doing the same for even numbers
for num in range(2, 20, 2):
    print(num)
# Prints: 2, 4, 6, 8, 10, 12, 14, 16, 18
# 20 is NOT printed, because the stop is exclusive
```

# range function

- By default range will use the value **1** for **step**, and **0** for **start**.  The **stop** must be provided as an argument.

- In any situation where you have a need for a sequence of numbers, generated in order, it's better to leverage the `range` function.

- This is why it's commonly used with the `for` statement.

# break and continue

There may be times where you want to leave a loop early. Or you may want to skip to the next iteration of the loop early.

In these cases Python gives us two commands, **break** and **continue**.

# break and continue

The **break** statement terminates the closest enclosing loop in which it appears.  In other words, we can leave the loop early.

The **continue** statement passes control to the next iteration of the enclosing **while** or **for** loop.  In other words, we can skip to the next run of the loop.

# break and continue

Below demonstrates usage of `break` and `continue`.

```
for i in range(6):
    if i == 3:
        break

print(i) # Prints out the value 3

text = ""

for i in range(10):
    if i == 3:
        continue
    text = text + str(i)

print(text) # Prints out 012456789
```

# References

Introduction to lists

https://docs.python.org/tutorial/introduction.html#lists

More on lists

https://docs.python.org/tutorial/datastructures.html#more-on-lists

# References

Introduction to the for statement

https://docs.python.org/reference/compound_stmts.html#for

# Demonstration:

- What is a list?
- Using the for statement to traverse a list
- range function
- break and continue