WELLCOME

PYTHON FOR DATA SCIENCE AND AI

YA MANON

SEQUENCE DATA TYPES

You can have data without information but you cannot have information without data.

-Daniel Keys Maran

SEQUENCE DATA TYPES

In this section we'll cover **sequence data types**, including lists, tuples, and ranges, which are capable of storing many values

TOPICS WE'LL COVER:

- List Basics
- List Operations
- Modifying Lists
- List Functions & Methods
- Nested Lists
- Copying Lists
- Tuples
- Ranges

GOALS FOR THIS SECTION:

- Learn to create lists and modify list elements
- Apply common list functions and methods
- Understand the different methods for copying lists
- Review the benefits of using tuples and ranges

LISTS

Lists are iterable data types capable of storing many individual elements

- List elements can be any data type, and can be added, changed, or removed at any time

```
random_list = ['snowboard', 10.54, None, True, -1]
```

Lists are created with square brackets **[]** *List elements are separated by commas*

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']
```

While lists can contain multiple data types, they often contain ***one data type***

```
empty_list = []

empty_list
```

```
[]
```

You can create an empty list and add elements later

```
list('Hello')
```

```
['H', 'e', 'l', 'l', 'o']
```

You can create lists from other iterable data types

MEMBERSHIP TESTS

Since lists are iterable data types, you can conduct **membership tests** on them

```
sizes_in_stock = ['XS', 'S', 'L', 'XL', 'XXL']

'M' in sizes_in_stock
```

```
False
```

*'M' is not an element in **sizes_in_stock**, so False is returned*

```
if 'M' in sizes_in_stock:
    print("I'll take the medium please.")
elif 'S' in sizes_in_stock:
    print("It'll be tight but small will do.")
else:
    print("I'll wait until you have medium.")
```

```
It'll be tight but small will do.
```

*'M' is not an element in **sizes_in_stock**, but 'S' is, so the print function under elif is executed*

LIST INDEXING

Elements in a list can be accessed via their **index**, or position within the list

- Remember that Python is 0-indexed, so the first element has an index of 0, not 1

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']

item_list[0]
```

```
'Snowboard'
```

An index of 0 grabs the first list element

```
item_list[3]
```

```
'Goggles'
```

An index of 3 grabs the fourth list element

LIST SLICING

Slice notation can also be used to access portions of lists [start: stop: stepsize]

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']

item_list[:3]
```

```
['Snowboard', 'Boots', 'Helmet']
```

list[:3] *grabs elements 0, 1, and 2*

Remember that 'stop' is non-inclusive!

```
item_list[::2]
```

```
['Snowboard', 'Helmet', 'Bindings']
```

list[::2] *grabs every other element in the list*

```
item_list[2:4]
```

```
['Helmet', 'Goggles']
```

list[2:4] *grabs the 3rd(index of 2) and 4th (index of 3) elements in the list*

UNPACKING LISTS

Lists elements can be **unpacked** into individual variables

```
item_list = ['Snowboard', 'Boots', 'Helmet']

s, b, h = item_list
print(s, b, h)
```

```
Snowboard Boots Helmet
```

s,***b*** *,and* ***h*** *are now string variables*

You need to specify the same number of variables as list elements or you will receive a **ValueError**

CHANGING LIST ELEMENTS

Lists elements can be **changed**, but not added, by using indexing

```
new_items = ['Coat', 'Backpack', 'Snowpants']

new_items[1] = 'Gloves'
new_items
```

```
['Coat', 'Gloves', 'Snowpants']
```

The second element in the list has changed from 'Backpack' to 'Gloves'

```
new_items = ['Coat', 'Backpack', 'Snowpants']

new_items[3] = 'Gloves'
new_items
```

```
IndexError: list assignment index out of range
```

The list only has 3 elements, so an index of 3 (the fourth element) does not exist

ADDING LIST ELEMENTS

You can **.append()** or **.insert()** a new element to a list

- **.append(*element*)** –adds an element to the end of the list
- **.insert(*index*, *element*)** –adds an element to the specified index in the list

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']

item_list.append('Coat')
print(item_list)
```

```
['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings', 'Coat']
```

'Coat' was added as the last element in the list

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']

item_list.insert(3, 'Coat')
item_list
```

```
['Snowboard', 'Boots', 'Helmet', 'Coat', 'Goggles', 'Bindings']
```

'Coat' was added as the fourth element in the list

COMBINING & REPEATING LISTS

Lists can be **combined**, or concatenated, with **+**and **repeated**with ***

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']
new_items = ['Coat', 'Backpack', 'Snowpants']

all_items = item_list + new_items
print(all_items)
```

```
['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings', 'Coat',
'Backpack', 'Snowpants']
```

```
new_items * 3
```

```
['Coat',
 'Backpack',
 'Snowpants',
 'Coat',
 'Backpack',
 'Snowpants',
 'Coat',
 'Backpack',
 'Snowpants']
```

REMOVING LIST ELEMENTS

There are two ways to **remove** lists elements:

1. The **del** keyword deletes the selected elements from the list

```
new_items = ['Coat', 'Backpack', 'Snowpants']

del new_items[1:3]

new_items
```

```
['Coat']
```

The second (index of 1) and third (index of 2) elements were deleted from the list

- **del**list_name(***slice***)

2. The **.remove()** method deletes the first occurrence of the specified value fromthe list

```
new_items = ['Coat', 'Backpack', 'Snowpants']

new_items.remove('Coat')
new_items
```

```
['Backpack', 'Snowpants']
```

'Coat' was removed from the list

- **.remove(*value*)**

LIST FUNCTIONS


```
transactions = [10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]
```

len(list_name) *Returns the number of elements in a list*

```
len(transactions)
```

```
6
```

sum(list_name) *Returns the sum of the elements in the list*

```
sum(transactions)
```

```
1483.88
```

min(list_name) *Returns the smallest element in the list*

```
min(transactions)
```

```
2.07
```

max(list_name) *Returns the largest element in the list*

```
max(transactions)
```

```
1242.66
```

SORTING LISTS

There are two ways to **sort** lists elements:

1. The **.sort()** method sorts the list permanently *(in place)*

```
transactions = [10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]

transactions.sort()
transactions
```

```
[2.07, 8.01, 10.44, 20.56, 200.14, 1242.66]
```

TIP: Don't sort in place until you're positive the code works as expected and you no longer need to preserve the original list

2. The **sorted** function returns a sorted list, but does not change the original *(not in place)*

```
transactions = [10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]
sorted(transactions)
```

```
[2.07, 8.01, 10.44, 20.56, 200.14, 1242.66]
```

```
transactions
```

```
[10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]
```

LIST METHODS


```
transactions = [10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]
```

.index(*value*)

Returns the index of a specified value within a list (returns -1 if not found)

```
transactions.index(200.14)
```

```
2
```

.count()

Counts the number of times a given value occurs in a list

```
transactions.count(200.14)
```

```
1
```

.reverse()

Reverses the order of the list elements in place

```
transactions.reverse()
transactions
```

```
[8.01, 2.07, 1242.66, 200.14, 20.56, 10.44]
```

PRO TIP: Use a negative slice to reverse the order “not in place”

NESTED LISTS

Lists stored as elements of another list are known as **nested lists**

```
list_of_lists = [['a', 'b', 'c'],
                 ['d', 'e', 'f'],
                 ['g', 'h', 'i']]
list_of_lists
```

```
[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
```

```
# grab the second element (a list)
list_of_lists[1]
```

```
['d', 'e', 'f']
```

Referencing a single index value will return an entire nested list

```
# grab the second element of the second element
list_of_lists[1][1]
```

```
'e'
```

Adding a second index value will return individual elements from nested lists

NESTED LISTS

List **methods**& **functions**still work with nested lists

```
list_of_lists = [['a', 'b', 'c'],
                 ['d', 'e', 'f'],
                 ['g', 'h', 'i']]
list_of_lists
```

```
[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
```

```
list_of_lists[1].append('k')
list_of_lists
```

'k' is being added as the last element to the second list

```
[['a', 'b', 'c'], ['d', 'e', 'f', 'k'], ['g', 'h', 'i']]
```

```
list_of_lists[2].count('h')
```

'h' appears once in the third list

```
1
```

COPYING LISTS

There are 3 different ways to **copy** lists:

1. **Variable assignment** –assigning a list to a new variable creates a “view”
   - *Any changes made to one of the lists will be reflected in the other*

2. **.copy()** –applying this method to a list creates a ‘shallow’ copy
   - *Changes to entire elements (nested lists) will not carry over between original and copy*
   - *Changes to individual elements within a nested list will still be reflected in both*

3. **deepcopy()** –using this function on a list creates entirely independent lists
   - *Any changes made to one of the lists will NOT impact the other*

deepcopyis overkill for the vast majority of uses cases, but worth being aware of

VARIABLE ASSIGNMENT

Copying a list via **variable assignment**creates a “view” of the list

- Both variables point to the same object in memory
- Changing an element in one list will result in a change to the other

```
list_of_lists = [['a', 'b', 'c'],
                 ['d', 'e', 'f'],
                 ['g', 'h', 'i']]

# copy list using variable assignment
list_of_lists2 = list_of_lists

# change element in original list
list_of_lists[1] = ['x', 'y', 'z']

# The change will be reflected in the copy made by assignment
list_of_lists2
```

```
[['a', 'b', 'c'], ['x', 'y', 'z'], ['g', 'h', 'i']]
```

*The change made to **list_of_lists** is reflected in **list_of_lists2***

COPY

Copying a list with the **.copy()** method creates a copy of the list

- Changes to entire elements (nested lists) will not carry over between original and
- copy Changes to individual elements within a nested list will still be reflected in both

```
list_of_lists = [['a', 'b', 'c'],
                 ['d', 'e', 'f'],
                 ['g', 'h', 'i']]

# use the copy method to create a copy
list_of_lists2 = list_of_lists.copy()

# replace the entire nested listed at index 0
list_of_lists[0] = ['x', 'y', 'z']

list_of_lists2
```

```
[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
```

Since the entire nested list at index 0 was replaced, the change is NOT reflected in the copy

```
# change the second element of the first nested list
list_of_lists[0][1] = 'Oh no!'

list_of_lists2
```

```
[['a', 'Oh no!', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
```

Since a single element (index of 1) within the nested list was modified, the change IS reflected in the copy

DEEPCOPY

Copying a list with the **deepcopy()** function creates a separate copy of the list

- Any changes made to one of the lists will NOT impact the other

This function is not part of base Python, so the copy library must be imported

```
from copy import deepcopy

list_of_lists = [['a', 'b', 'c'],
                 ['d', 'e', 'f'],
                 ['g', 'h', 'i']]

list_of_lists2 = deepcopy(list_of_lists)

list_of_lists[0][1] = 'Oh no!'

list_of_lists2
```

```
[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
```

The change is NOT reflected in the copy, even though a single element (index of 1) within the nested list was modified

TUPLES

Tuples are iterable data types capable of storing many individual items

- Tuples are very similar to lists, except they are **immutable**
- Tuple items can still be any data type, but they CANNOT be added, changed, or removed once the tuple is created

```
item_tuple = ('Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings')

item_tuple
```

```
('Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings')
```

*Tuples are created with parenthesis **()**, or the tuple() function*

```
item_tuple[3]
```

```
'Goggles'
```

```
item_tuple[:3]
```

```
('Snowboard', 'Boots', 'Helmet')
```

```
len(item_tuple)
```

```
5
```

List operations that don't modify their elements work with tuples as well

WHY USE TUPLES?

1. Tuples require **less memory** than a list

```
import sys

item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']
item_tuple = ('Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings')

print(f"The list is {sys.getsizeof(item_list)} " + 'Bytes')
print(f"The tuple is {sys.getsizeof(item_tuple)} " + 'Bytes')
```

```
The list is 120 Bytes
The tuple is 80 Bytes
```

*The tuple is **33% smaller** than the list*

For heavy data processing, this can be a big difference

2. Operations **execute quicker** on tuples than on lists

```
import timeit

# calculate time of summing list 10000 times
print(timeit.timeit("sum([10.44, 20.56, 200.14, 1242.66, 2.07, 8.01])",
                    number=10000))

# calculate time of summing tuple 10000 times
print(timeit.timeit("sum((10.44, 20.56, 200.14, 1242.66, 2.07, 8.01))",
                    number=10000))
```

```
0.0076123750004626345
0.003229583000575076
```

*The tuple executed over **twice as fast** as the list*

WHY USE TUPLES?

3. Tuples **reduce user error** by preventing modification to data
    - There are cases in which you explicitly do not want others to be able to modify data

4. Tuples are common output in **imported functions**

```
a = 1
b = 2
c = 3

a, b, c
```

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

Even asking to return multiple variables in a code cell returns them as a tuple

RANGES

Ranges are sequences of integers generated by a given start, stop, and step size

- They are more memory efficient than tuples
- They save time, as you don't need to write the list of integers manually in the code
- They are commonly used with loops *(more on that later!)*

```
example_range = range(1, 5, 1)

print(example_range)
```

```
range(1, 5)
```

Note that printing a range does NOT return the integers

```
print(list(example_range))
```

```
[1, 2, 3, 4]
```

```
print(tuple(range(len('Hello'))))
```

```
(0, 1, 2, 3, 4)
```

You can retrieve them by converting to a list or tuple

KEY TAKEAWAYS

- Lists and tuples are sequence data types capable of storing many values
  - *Lists allow you to add, remove, and change elements, while tuples do not*
- Use**indexing**and **slicing**to access & modify list elements
  - *Make sure you are comfortable with this syntax, as you will use it frequently in data analysis libraries as well*
- Built-in **functions** and**methods** make working with sequences easier
  - *All the list functions covered work withtuples as well, and tuples have many identical methods, like index() and count(). Takea minute to skim through the available methods in the Python documentation.*
- Ranges help create a **sequence of integers** efficiently
  - *These will become particularly powerful when combined with loops*