Week 3: Thursday

Collections

Data Collections

What are Data Collections?

- Data collections are a way to store multiple data items in a single variable.
- We have already met the basics of these data collections

Lists

Lists, a reminder

- lists are ordered collections of items
- lists are mutable
- lists are created using square brackets
- lists can contain any type of data
  - This is important
  - lists can be nested

Declaring and initializing a list

- Python is dynamically typed == typed at runtime
- Python is strongly typed == once inferred, python treats them that way.
  - unless we explicitly change the type via type casting.

- TLDR: `l = []` is all we need to do to declare and initialize a list.
- It will always be a list until we cast it to another type.

Lists can be multidimensional

- 1 dimensional list `l = [1, 2, 3, 4, 5]`
- 2 dimensional list `l = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`
- 3 dimensional list `l = [[[1, 2, 3], [4, 5, 6], [7, 8, 9]], [[1, 2, 3], [4, 5, 6], [7, 8, 9]], [[1, 2, 3], [4, 5, 6], [7, 8, 9]]]`

Okay that looks stupid. Let's make it more readable.

```
d1 = [1, 2, 3]
d2 = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

3D list

```
d3 = [
  [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
  ],
  [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
  ],
  [
    [1, 2, 3],
    [4, 5, 6],
```

```
    [7, 8, 9]
  ]
]
```

When do we use multidimensional lists?

- ALL THE TIME!!!!
- Representing space two-dimensionally
  - think a chessboard

Chessboard

```
chess = [
  ['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R'],
  ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
  ['','','','','','','','',],
  ['','','','','','','','',],
  ['','','','','','','','',],
  ['','','','','','','','',],
  ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
  ['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R'],
]
for i in chess:
  print(i)
```

```
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
```

When do we use multidimensional lists?

- Representing matrices.
- Representing a collection of objects that have multiple attributes.
- A Dataframe (what we want to get to) is a multidimensional list.

A Dataframe, we will get more into this later.

CSV

```
Name, Age, Gender
Nathan, 38, M
Sally, 25, F
Rebecca, 39, F
```

Dataframe

```
dataframe = [
  ["Nathan", 38, "M"],
  ["Sally", 25, "F"],
  ["Rebecca", 39, "F"]
]
```

Selecting elements in a list

- How do we select elements from a list?
- With index notation.
- `l[0]` will return the first element in the list.

```
l = [1, 2, 3, 4, 5]
print(l[0]) # returns 1
```

```
1
```

Changing elements in a list

- use typical assignment notation
- `l[0] = 10` will change the first element in the list to 10.

```
l[0] = 10
print(l)
```

```
[10, 2, 3, 4, 5]
```

Selecting complicated elements

- Remember our chessboard?
- Let's get each row individually.

```
print(chess[0])
print(chess[1])
print(chess[2])
print(chess[3])
print(chess[4])
print(chess[5])
print(chess[6])
print(chess[7])
```

```
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
```

Or we can get the value of each square

- Using Index notation
- `chess[0][0]` will return the value of the first square on the first row.

```
print(chess[0][0]) # returns 'R'
print(chess[1][0]) # returns 'P'
```

```
R
P
```

Maybe We Should Play chess

```
for i in chess:
    print(i)
```

```
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
```

Chess Continued (redefining items)

First Move

```
# P1 Move 1
chess[2][2] = chess[1][2]  # Move pawn.
chess[1][2] = ""  # remove vacated location

# print(chess[1][2])
for i in chess:
    print(i)
```

```
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
['P', 'P', '', 'P', 'P', 'P', 'P', 'P']
['', '', 'P', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
```

Chess continued

Second Move

```
# P2 Move 1
chess[5][2] = "N"  # Move Knight.
chess[7][1] = ""  # remove vacated location

for i in chess:
    print(i)
```

```
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
['P', 'P', '', 'P', 'P', 'P', 'P', 'P']
['', '', 'P', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', 'N', '', '', '', '', '']
['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
['R', '', 'B', 'Q', 'K', 'B', 'N', 'R']
```

Chess Again

- Wait, we can do this better.

```
pawns = ["P"] * 8
goodens = ["R", "N", "B", "Q", "K", "B", "N", "R"]
empty = [""] * 8

chess = [goodens, pawns, empty, empty, empty, empty, pawns, goodens]
```

```
for i in chess:
    print(i)
```

```
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
```

Chess Continued (redefining items)

First Move

```
# P1 Move 1
chess[2][2] = chess[1][2]  # Move pawn.
chess[1][2] = ""  # remove vacated location

# print(chess[1][2])
for i in chess:
    print(i)
```

```
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
['P', 'P', '', 'P', 'P', 'P', 'P', 'P']
['', '', 'P', '', '', '', '', '']
['', '', 'P', '', '', '', '', '']
['', '', 'P', '', '', '', '', '']
['', '', 'P', '', '', '', '', '']
['P', 'P', '', 'P', 'P', 'P', 'P', 'P']
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
```

Chess continued

Second Move

```
# P2 Move 1
chess[5][2] = "N"  # Move Knight.
chess[7][1] = ""  # remove vacated location

for i in chess:
    print(i)
```

```
['R', '', 'B', 'Q', 'K', 'B', 'N', 'R']
['P', 'P', '', 'P', 'P', 'P', 'P', 'P']
['', '', 'N', '', '', '', '', '']
['', '', 'N', '', '', '', '', '']
['', '', 'N', '', '', '', '', '']
['', '', 'N', '', '', '', '', '']
['P', 'P', '', 'P', 'P', 'P', 'P', 'P']
['R', '', 'B', 'Q', 'K', 'B', 'N', 'R']
```

What's Happening?

```
for i in chess:
    print(i)
```

```
['R', '', 'B', 'Q', 'K', 'B', 'N', 'R']
['P', 'P', '', 'P', 'P', 'P', 'P', 'P']
['', '', 'N', '', '', '', '', '']
['', '', 'N', '', '', '', '', '']
['', '', 'N', '', '', '', '', '']
['', '', 'N', '', '', '', '', '']
['P', 'P', '', 'P', 'P', 'P', 'P', 'P']
['R', '', 'B', 'Q', 'K', 'B', 'N', 'R']
```

Solving the problem

```
pawns = ["P"] * 8
goodens = ["R", "N", "B", "Q", "K", "B", "N", "R"]
empty = [""] * 8
```

```
chess = [goodens.copy(), pawns.copy(), empty.copy(), empty.copy(), empty.copy(), empty.copy(),
for i in chess:
    print(i)
```

```
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
```

Now it works

```
# P2 Move 1
chess[5][2] = "N"  # Move Knight.
chess[7][1] = ""  # remove vacated location

for i in chess:
    print(i)
```

```
['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '']
['', '', 'N', '', '', '', '', '']
['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
['R', '', 'B', 'Q', 'K', 'B', 'N', 'R']
```

Other list methods

method	explanation
`append()`	adds an element to the end of the list
`extend()`	adds all elements of a list to the another list

method	explanation
`insert()`	inserts an item at the defined index

Other list methods, cont.

|`remove()`|removes an item from the list| |`pop()`|removes and returns an element at the given index and returns the item| |`clear()`|removes all items from the list| |`index()`|returns the index of the first matched item|

Examples

```
l = [1, 2, 3, 4, 5]
l.append(6)
print("append: ", l)
l.extend([7, 8, 9])
print("extend: ", l)
l.insert(0, 0)
print("insert: ", l)
l.remove(4)  # removes the first 4, not the index 4
print("remove: ", l)
```

```
append:  [1, 2, 3, 4, 5, 6]
extend:  [1, 2, 3, 4, 5, 6, 7, 8, 9]
insert:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
remove:  [0, 1, 2, 3, 5, 6, 7, 8, 9]
```

Examples Continued

```
print(l)
print("remove: ", l)
l.pop(0)  # removes the first element
print("pop: ", l)
l.pop()  # removes the last element
print("pop: ", l)
```

```
mi = l.index(3)  # returns the index of 3
print("index: ", l)
l.clear()  # removes all elements
print("clear: ", l)
```

```
[0, 1, 2, 3, 5, 6, 7, 8, 9]
remove:  [0, 1, 2, 3, 5, 6, 7, 8, 9]
pop:  [1, 2, 3, 5, 6, 7, 8, 9]
pop:  [1, 2, 3, 5, 6, 7, 8]
index:  [1, 2, 3, 5, 6, 7, 8]
clear:  []
```

Dictionaries

Declaring/Initializing a dictionary

- Dictionaries are unordered collections of items
- Dictionaries are mutable
- Dictionaries are created using curly braces
- Dictionaries contain key-value pairs
- `d = {}` is all we need to declare and initialize a dictionary.
- or `d = dict()`

Declaring and reading a dictionary

```
people = {"name": "Nathan", "age": 38, "is_happy": True}
print("name is ", people["name"])
print("age is ", people["age"])
```

```
name is  Nathan
age is  38
```

Selecting Specific Items

- We can get the `keys()`: returns a list of all keys
- We can get the `values()`: returns a list of all values
- We can get the `items()`: returns a list of key-value pairs in tuple format

```
people = {"name": "Nathan", "age": 38, "is_happy": True}
print("keys: ", people.keys())
print("values: ", people.values())
print("items: ", people.items())
```

```
keys:  dict_keys(['name', 'age', 'is_happy'])
values:  dict_values(['Nathan', 38, True])
items:  dict_items([('name', 'Nathan'), ('age', 38), ('is_happy', True)])
```

Selecting Specific Items

- Guess what? We use index notation on that items() list of tuplse

```
print(people)
itm = list(people.items())  # convert to list
print(itm[0][1])  # returns the value of the first key-value pair
```

```
{'name': 'Nathan', 'age': 38, 'is_happy': True}
Nathan
```

Changing Items

- Changing items uses assignment notation. (same as a list)

```
people = {"name": "Nathan", "age": 38, "is_happy": True}
people["name"] = "Sally"

print("name is , ", people["name"])
```

```
name is ,  Sally
```

Changing Items, part II

- We can also update items using the `update()` method
- But we can update multiple items at once.

```
people = {"name": "Nathan", "age": 38, "is_happy": True}
people.update({"name": "Sally", "age": 25})
print(people)
```

```
{'name': 'Sally', 'age': 25, 'is_happy': True}
```

Which is better?

- It depends on what you are doing.
- If you are changing one item, use assignment notation. (faster)
- If you are changing multiple items, use the `update()` method.

How do we know?

- We `%timeit()`

```
people = {"name": "Nathan", "age": 38, "is_happy": True}
%timeit people["name"] = "Sally"
%timeit people.update({"name": "Johnathan"})
```

```
27.2 ns ± 3.33 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
111 ns ± 2.47 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

- `%timeit()` calculates the average time taken to run the code over multiple runs. (like millions)

Adding Items

- We can add items with the same notation
- We just use a key that doesn't exist.
- But we can also add two dictionaries together with update and `**` operator.

```
people = {"name": "Nathan", "age": 38, "is_happy": True}
more_people = {"lastname": "Sally", "months": 25, "is_sad": False}
people["another_key"] = "another_value"
print("1", people)
me = {"birthday": "June 1st"}

people.update(more_people)
print("2", people)

final = {**people, **me}
for i in final.items():
    print(i)
```

```
1 {'name': 'Nathan', 'age': 38, 'is_happy': True, 'another_key': 'another_value'}
2 {'name': 'Nathan', 'age': 38, 'is_happy': True, 'another_key': 'another_value', 'lastname':
('name', 'Nathan')
('age', 38)
('is_happy', True)
('another_key', 'another_value')
('lastname', 'Sally')
('months', 25)
('is_sad', False)
('birthday', 'June 1st')
```

Removing Items

- We can remove items with the `pop()` method

```
people = {"name": "Nathan", "age": 38, "is_happy": True}
j = people.pop("name")
print("value of j", j)
print("value of people:", people)
people.pop("age")
print("value of people after 2nd pop: ", people)
```

```
value of j Nathan
value of people: {'age': 38, 'is_happy': True}
value of people after 2nd pop:  {'is_happy': True}
```

Other Dictionary Methods

Method	Explanation
`.clear()`	removes all items from the dictionary
`.copy()`	returns a shallow copy of the dictionary
`.fromkeys()`	returns a dictionary with the specified keys and values
`.get()`	returns the value of the specified key

Other Dictionary Methods, cont.

Method	Explanation
`.items()`	returns a list containing a tuple for each key value pair
`.keys()`	returns a list containing the dictionary's keys
`.popitem()`	removes the last inserted key-value pair
`.values()`	returns a list of all the values in the dictionary