Homework 1

You will turn in this notebook without any of the data files. All cells must be executed and the notebook saved with output included. You must also save your fully executed notebook as a pdf following the instructions in the Student Handbook posted on Canvas.

All of the **Problem** cells serve as markers. We will use them to divide all of the submitted notebooks into new notebooks that contain ONLY answers to one problem at a time. **Do not move or edit the Problem cells**. Always put your answers in the provided code or markdown (text) cells.

Debugging tip: Python error messages are terrible! It can be scary and confusing to see hundreds of lines of errors pop up. **Most of the error is just *smoke*. Ignore it and scroll to the bottom to find the *fire*.** Try googling the relevant parts of the error message to debug.

If you get stuck for more than 30 minutes on one problem when working alone, ask a TA or post on Ed Discussion. If you are working with others, ask if you get stuck for more than 15 minutes.

If you use any sources aside from the standard documentation of Python libraries, you must cite the sources (e.g., StackOverflow links) used to answer the below questions. If you use ChatGPT or another AI system, you must include proof of the prompt you used, as well as a line-by-line explanation of why the code is correct. Simply noting that the code makes sense does not suffice as an explanation.

Overall, Homework 1 consists of two sections to facilitate practice using NumPy, Pandas, and SQL. The goal of section 1 is to practice basic NumPy skills necessary this semester. Section 2 focuses on using basic Pandas and SQL skills necessary this semester.

NetID: MNS66

Problem 0

Do not move or modify this cell, we use it for automated homework analysis.

This problem is for you to use to cite your sources.

You will update this problem over the course of the homework. Describe problems, error messages, and bugs you encountered, and how you fixed or addressed them. List URLs for any online resources (like Stack Overflow) you found useful and mention fellow students, TAs, or other people who helped you. If you use ChatGPT or other AI, you must include proof of the prompt you used, as well as an explanation of why the resulting

code is correct. If you do not appropriately fill out Problem 0, you will receive 0 points on the entire homework assignment.

ADD YOUR SOURCES HERE

Section 0: Import Packages

NumPy is a commonly used Python package for working with data. Below, we will practice using arrays and indexing.

Whenever you use NumPy, you first need to import it with the following line of code.

Run the following block of code by pressing the `shift` and `enter` keys simultaneously. This is the general keyboard shortcut for running a block of code in a notebook like this.

In [1]:
```
import numpy as np
```

Whenever you use pandas, you first need to import it with the following line of code. We'll also be using SQL via the duckdb package; you can see more documentation *here*. Make sure you can also import duckdb:

In [2]:
```
import pandas as pd
import duckdb
```

At the beginning of class, we told you about DataFrames. They have named columns and one datatype per column. The pandas library is our main support for DataFrames in python. This homework will teach you standard operations on DataFrames with pandas and how they compare to SQL.

Note: Remember that the order in which you run cells matters! If you overwrite the dataframe `df` and then try to run an earlier cell, you may get errors. One way to make sure you're not breaking things is to clear outputs of all cells and then Run All from the beginning.

Note: In a Jupyter notebook like this, the value in the last line of a code cell will be printed, even without a `print()` statement. We have you use `print()` statements in every exercise because it will allow you to print multiple values from a cell when you need to.

The exercises are structured as such: We may provide some given code (such as initializing arrays) that you will need to execute. Then, we will list out a series of exercises to complete. These are followed with a blank code block for you to fill in with

your code for each exercise. Finally, we will provide the expected outputs so you can check that your code has returned the correct answer.

Section 1: NumPy Skills

A. NumPy Arrays (15 points)

The basic datatype in NumPy is an *array*, also called an *ndarray*. You can think of an array as a list of numbers. A basic way to create an array is to pass a Python list to the `np.array()` function.

Goal: Print an array and access the number of dimensions of an array.

Python's `print()` function displays the value of a variable.

Arrays have *axes*, also called *dimensions*. A Python list has one dimension; you can specify an element of a list with one index number. A matrix has two dimensions because in order to specify an element of a matrix, you need two numbers. The numbers specify the row and column.

In [3]:
```
# Exercises A1 – A6 Given code
A = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])
B = np.array([[0, 1, 2, 3, 4, 5, 6, 7, 8]])
C = np.array([[0],
              [1],
              [2],
              [3],
              [4],
              [5],
              [6],
              [7],
              [8]])
D = np.array([[ 0,  1,  2,  3,  4,  5,  6,  7, 8],
              [ 9, 10, 11, 12, 13, 14, 15, 16, 17]])
```

Exercise A1. Print array A

Exercise A2. Print array B

Exercise A3. Print array C

Exercise A4. Print the number of dimensions of array A using the array's `.ndim` attribute.

Exercise A5. Print the number of dimensions of array B.

Exercise A6. Print the number of dimensions of array D.

Exercise A7. Use the `np.array()` function to create any array with four axes/dimensions. Print the number of dimensions of the array.

In [4]:
```
# A1: your code here
print(A)

# A2: your code here
print(B)

# A3: your code here
print(C)

# A4: your code here
print(f"Num dimensions of A: {A.ndim}")

# A5: your code here
print(f"Num dimensions of B: {B.ndim}")

# A6: your code here
print(f"Num dimensions of D: {D.ndim}")

# A7: your code here
my_list = [[[[1,2]], [[3,4]], [[5,6]], [[7,8]], [[9,10]], [[11,12]]]]
arr = np.array(my_list)
print(f"Num dimensions of arr: {arr.ndim}")
```

```
[0 1 2 3 4 5 6 7 8]
[[0 1 2 3 4 5 6 7 8]]
[[0]
 [1]
 [2]
 [3]
 [4]
 [5]
 [6]
 [7]
 [8]]
Num dimensions of A: 1
Num dimensions of B: 2
Num dimensions of D: 2
Num dimensions of arr: 4
```

Expected A1-A7 Output

```
[0 1 2 3 4 5 6 7 8]
[[0 1 2 3 4 5 6 7 8]]
[[0]
 [1]
 [2]
 [3]
 [4]
 [5]
```

```
 [6]
 [7]
 [8]]
1
2
2
4
```

Goal: Get the shape of an array.

An array's *shape* is a tuple representing the size of each dimension of an array. For a 2d array, the elements of the tuple correspond to height and width.

In [5]:
```
# Exercise A12 Given code
E = np.array([[0, 1, 2],
              [3, 4, 5],
              [6, 7, 8]])
```

Exercise A8. Print the shape of array D using the array's `.shape` attribute.

Exercise A9. Print the shape of array A

Exercise A10. Print the shape of array B.

Exercise A11. Print the shape of array C.

Exercise A12. Print the shape of array E.

In [6]:
```
# A8: your code here
print(f"Shape of D: {D.shape}")

# A9: your code here
print(f"Shape of A: {A.shape}")

# A10: your code here
print(f"Shape of B: {B.shape}")

# A11: your code here
print(f"Shape of C: {C.shape}")

# A12: your code here
print(f"Shape of E: {E.shape}")
```

```
Shape of D: (2, 9)
Shape of A: (9,)
Shape of B: (1, 9)
Shape of C: (9, 1)
Shape of E: (3, 3)
```

Expected A8-A12 Output

```
(2, 9)
(9,)
(1, 9)
(9, 1)
(3, 3)
```

Goal: Reshape an array and get the number of elements in an array.

A NumPy array's `.reshape()` method takes the values in the array and rearranges them. The part that takes some getting used to is which values go in which new positions.

The `np.arange()` function creates arrays with integers in increasing order. For example, the array `[0 1 2 3 4 5 6 7 8]` from the previous exercises can also be generated by the line `np.arange(9)`.

In [7]:
```
# Exercise A13 Given code
F = np.arange(18)
```

Exercise A13. Modify array F to have the same elements in two rows of equal size (equivalent to previously-defined array D). Print the resulting array.

Exercise A14. Recreate array B using array A and the reshape function. Print the resulting array.

Exercise A15. Recreate array E using array A and the reshape function. Print the resulting array.

Exercise A16. Use an array's `.size` attribute to print the number of elements in each of arrays A, B, E.

In [8]:
```
# A13: your code here
F = F.reshape(2,9)
print(F)

# A14: your code here
B_new = A.reshape(1,9)
print(B_new)

# A15: your code here
E_new = A.reshape(3,3)
print(E_new)

# A16: your code here
print(f"Size of A: {A.size}")
print(f"Size of B: {B.size}")
print(f"Size of E: {E.size}")
```

```
[[ 0  1  2  3  4  5  6  7  8]
 [ 9 10 11 12 13 14 15 16 17]]
[[0 1 2 3 4 5 6 7 8]]
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Size of A: 9
Size of B: 9
Size of E: 9
```

Expected A13 - A16 Output:

```
[[ 0  1  2  3  4  5  6  7 8]
 [ 9 10 11 12 13 14 15 16 17]]
[[0 1 2 3 4 5 6 7 8]]
[[0 1 2]
 [3 4 5]
 [6 7 8]]
9
9
9
```

Goal: Get the Python type of NumPy arrays, and get the type of the elements held in a NumPy array.

You can get the type of a variable in Python with the built-in `type()` function. NumPy arrays are a single type that can hold elements of another type. For example, you could have an array of integers or an array of strings.

In [9]:
```
# Exercise A19 - A20 Given code
G = np.array(['maple', 'london plane', 'beech', 'white oak'])
```

Exercise A17. Print the Python type of array E by using Python's built-in `type()` function.

Exercise A18. Print the NumPy array's `dtype` attribute in order to see the type of elements that array E holds. (Older operating systems may return `int32`.)

Exercise A19. Print the Python type of array G.

Exercise A20. Print the NumPy array's `dtype` attribute of array G.

In [10]:
```
# A17: your code here
print(type(E))

# A18: your code here
print(E.dtype)

# A19: your code here
```

```
print(type(G))

# A20: your code here
print(G.dtype)
```

```
<class 'numpy.ndarray'>
int64
<class 'numpy.ndarray'>
<U12
```

Expected A17 - A20 Output

```
<class 'numpy.ndarray'>
int64
<class 'numpy.ndarray'>
<U12
```

(The last one is a string datatype, as described here)

Goal: Create an array. Understand the difference between `np.empty()` and `np.zeros()`.

The `np.empty()` function creates an array with no guarantees on the values of elements inside it. If you want to use an array created in this way, you need to explicitly set the values.

The `np.zeros()` function creates an array where every entry is 0. This is useful when you want to immediately use the values in the array, or if the default value of an entry should be 0.

In [11]:
```
# Exercises A24 – A25 Given code
l = [14, 12, 10, 8, 6, 4, 2, 0]

list1 = [10, 8, 6, 4, 2, 0]
list2 = [10.5, 8.5, 6.5, 4.5, 2.5, 0.5]
list3 = [11, 9, 7, 5, 3, 1]
```

Exercise A21. Create a 2x3 array using the `np.empty()` function. Print the array.

Exercise A22. Create a 4x3 array using the `np.zeros()` function. Print the array.

Exercise A23. Evenly spaced numbers are often useful for plotting. Create a 1d array of 12 equally spaced numbers between 0 and 1 (inclusive) using the `np.linspace()` function. Print the array.

Exercise A24. Create a NumPy array from list l using the `np.array()` function. Reshape the array to be 4x2. Print the array.

Exercise A25. Create a 3x6 NumPy array from the lists 1-3, where the elements of `list1` are in the first row, the elements of `list2` are in the second row and the elements of `list3` are in the third row. Print the array.

In [12]:
```
# A21: your code here
two_by_three = np.empty((2,3))
print(two_by_three)

# A22: your code here
four_by_three = np.zeros((4,3))
print(four_by_three)

# A23: your code here
plotting_arr = np.linspace(0, 1, 12)
print(plotting_arr)

# A24: your code here
l_arr = np.array(l)
l_arr = l_arr.reshape(4,2)
print(l_arr)


# A25: your code here
three_by_six = np.array([list1, list2, list3])
print(three_by_six)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
[0.         0.09090909 0.18181818 0.27272727 0.36363636 0.45454545
 0.54545455 0.63636364 0.72727273 0.81818182 0.90909091 1.        ]
[[14 12]
 [10  8]
 [ 6  4]
 [ 2  0]]
[[10.   8.   6.   4.   2.   0. ]
 [10.5  8.5  6.5  4.5  2.5  0.5]
 [11.   9.   7.   5.   3.   1. ]]
```

Expected A21 - A25 Output

Note for A21: This is just a representative output. Due to the nature of `np.empty()`, you may have a slightly different output. This is expected.

```
[[0.e+000 2.e-323 0.e+000]
 [0.e+000 0.e+000 0.e+000]]
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
[0.         0.09090909 0.18181818 0.27272727 0.36363636
```

```
0.45454545
 0.54545455 0.63636364 0.72727273 0.81818182 0.90909091 1.
]
[[14 12]
 [10  8]
 [ 6  4]
 [ 2  0]]
[[10.   8.   6.   4.   2.   0. ]
 [10.5  8.5  6.5  4.5  2.5  0.5]
 [11.   9.   7.   5.   3.   1. ]]
```

B. Indexing (15 points)

Goal: Get and set the value of a single entry in an array.

In order to get an element from an array, you *index* the array with the element's position along each dimension/axis. The positions are numbered starting at 0.

You can use square bracket syntax like with a Python list. Different dimensions are separated with commas.

You might find the NumPy reference for indexing helpful.

You can set the value of an entry in an array using the same bracket syntax used for indexing. For example, `A[i, j] = 1` sets the entry in row `i`, column `j` to 1.

In [13]:
```
# Exercises B1 – B6 Given code
H = np.arange(24).reshape((3, 8))
```

Exercise B1. Print the top-left entry in array H, which is the entry in the first row, first column.

Exercise B2. Print the entry in the first row, fifth column.

Exercise B3. Print the entry in the second row, fourth column.

Exercise B4. Print the entry `22` using indexing.

Exercise B5. Use negative indexing to print the bottom right element of array H, which is the entry in the last row, last column.

Exercise B6. Set the entry in the third row, fourth column to 200. Print the array.

In [14]:
```
# B1: your code here
print(H[0, 0])

# B2: your code here
print(H[0, 4])
```

```
# B3: your code here
print(H[1, 3])

# B4: your code here
print(H[2, 6])

# B5: your code here
print(H[-1, -1])

# B6: your code here
H[2, 3] = 200
print(H)
```

```
0
4
11
22
23
[[  0   1   2   3   4   5   6   7]
 [  8   9  10  11  12  13  14  15]
 [ 16  17  18 200  20  21  22  23]]
```

Expected B1 - B6 Output:

```
0
4
11
22
23
[[  0   1   2   3   4   5   6   7]
 [  8   9  10  11  12  13  14  15]
 [ 16  17  18 200  20  21  22  23]]
```

Goal: Use slicing to access parts of an array.

Slicing allows you to access multiple elements of an array at the same time. A slice is defined using a colon. For example:

- `A[:, 1]` accesses all of the entries in column 1 of `A`.
- `A[0, :]` accesses all of the entries in row 0 of `A`.
- `A[:, 1:3]` accesses all of the entries in columns 1 (inclusive) to 3 (not inclusive) of `A` (i.e. columns 1 and 2).

Syntax is described here.

In [15]:
```
# Exercise B7 - B10 Given code
M = np.arange(50)
H = np.arange(24).reshape((3, 8))
```

Exercise B7. Use slicing to print the first 10 elements of array M.

Exercise B8. Use slicing to print the thirty elements of array M from index 15 to index 35 (inclusive).

Exercise B9. Use negative indexing and slicing to print the last five elements of array M.

Exercise B10. Use slicing and negative indexing to print the last column of array H.

In [16]:
```
# B7: your code here
print(M[:10])

# B8: your code here
print(M[15:36])

# B9: your code here
print(M[-5:])

# B10: your code here
print(H[:, -1])
```

```
[0 1 2 3 4 5 6 7 8 9]
[15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35]
[45 46 47 48 49]
[ 7 15 23]
```

Expected B7 - B10 Output:

```
[0 1 2 3 4 5 6 7 8 9]
[15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35]
[45 46 47 48 49]
[ 7 15 23]
```

Goal: Get and set the value of an entire row or column of a 2D array

In [17]:
```
# B11 - B14 Given code
H = np.arange(24).reshape((3, 8))
```

Exercise B11. Print all of the entries in the fourth column of H.

Exercise B12. Print all of the entries in the second row of H.

Exercise B13. Set all of the entries in the first colum of H to 1. Print the array.

Exercise B14. Create a 2x2 array that contains the four entries located in a) the second and third rows and b) the last two columns. Print this array.

In [18]:
```
# B11: your code here
print(H[:, 3])
```

```
# B12: your code here
print(H[1, :])

# B13: your code here
H[:, 0] = 1
print(H)

# B14: your code here
row_one, row_two = H[-2, -2:], H[-1, -2:]
two_by_two = np.array([row_one, row_two])
print(two_by_two)
```

```
[ 3 11 19]
[ 8  9 10 11 12 13 14 15]
[[ 1  1  2  3  4  5  6  7]
 [ 1  9 10 11 12 13 14 15]
 [ 1 17 18 19 20 21 22 23]]
[[14 15]
 [22 23]]
```

Expected B11 - B14 Output:

```
[ 3 11 19]
[ 8  9 10 11 12 13 14 15]
[[ 1  1  2  3  4  5  6  7]
 [ 1  9 10 11 12 13 14 15]
 [ 1 17 18 19 20 21 22 23]]
[[14 15]
 [22 23]]
```

Goal: Use slicing to print a subset of a large array.

Sometimes using the built-in Python `print()` function doesn't print all the values of a large array.

In [19]:
```
# B15 Given code
N = np.arange(10000).reshape((100,100))
```

Exercise B15. Use slicing to print the full tenth column of the array.

In [20]:
```
# B15: your code here
print(N[:, 9])
```

```
[   9  109  209  309  409  509  609  709  809  909 1009 1109 1209 1309
 1409 1509 1609 1709 1809 1909 2009 2109 2209 2309 2409 2509 2609 2709
 2809 2909 3009 3109 3209 3309 3409 3509 3609 3709 3809 3909 4009 4109
 4209 4309 4409 4509 4609 4709 4809 4909 5009 5109 5209 5309 5409 5509
 5609 5709 5809 5909 6009 6109 6209 6309 6409 6509 6609 6709 6809 6909
 7009 7109 7209 7309 7409 7509 7609 7709 7809 7909 8009 8109 8209 8309
 8409 8509 8609 8709 8809 8909 9009 9109 9209 9309 9409 9509 9609 9709
 9809 9909]
```

Expected B15 Output:

```
[   9  109  209  309  409  509  609  709  809  909 1009 1109
1209 1309
 1409 1509 1609 1709 1809 1909 2009 2109 2209 2309 2409 2509
2609 2709
 2809 2909 3009 3109 3209 3309 3409 3509 3609 3709 3809 3909
4009 4109
 4209 4309 4409 4509 4609 4709 4809 4909 5009 5109 5209 5309
5409 5509
 5609 5709 5809 5909 6009 6109 6209 6309 6409 6509 6609 6709
6809 6909
 7009 7109 7209 7309 7409 7509 7609 7709 7809 7909 8009 8109
8209 8309
 8409 8509 8609 8709 8809 8909 9009 9109 9209 9309 9409 9509
9609 9709
 9809 9909]
```

Goal: use boolean indexing to access specific elements

Boolean indexing, also called *logical indexing*, allows you to select elements of an array that meet certain criteria. There is a general pattern of starting with an array, creating a boolean array where `True` corresponds to elements that satisfy a predicate, and then passing this boolean array as an index back to the original array to select those elements.

The NumPy indexing reference contains a section on boolean indexing. The online resource Python Like You Mean It also has a helpful section about boolean indexing.

In [21]:
```
# B16 - B18 Given code
P = np.arange(36).reshape((6, 6))
```

Exercise B16. Create an 6x6 array called `P_bool` which contains `True` if the corresponding entry in `P` is odd and `False` otherwise. More concretely: a) `P_bool[i, j] = True` if the integer in `P[i, j]` is odd, and b) `P_bool[i, j] = False` if the integer in `P[i, j]` is even. Print `P_bool` once you have created it.

Hint: the modulo operator in Python yields the remainder when dividing. For example, `4 % 2` yields 0 because 4 is even, and `5 % 2` yields 1 because 5 is odd. The expression `4 % 2 == 0` would therefore yield `True`, while `5 % 2 == 0` would yield `False`.

Exercise B17. Print all the odd entries of the array P using one line of logical indexing.

Exercise B18. Set all the even entries of the array P to 1 using one line of logical indexing. Print the resulting array.

In [22]:
```
# B16: your code here
P_bool = P % 2 == 1
print(P_bool)

# B17: your code here
print(P[P % 2 == 1])

# B18: your code here
P[P % 2 == 0] = 1
print(P)
```

```
[[False  True False  True False  True]
 [False  True False  True False  True]
 [False  True False  True False  True]
 [False  True False  True False  True]
 [False  True False  True False  True]
 [False  True False  True False  True]]
[ 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35]
[[ 1  1  1  3  1  5]
 [ 1  7  1  9  1 11]
 [ 1 13  1 15  1 17]
 [ 1 19  1 21  1 23]
 [ 1 25  1 27  1 29]
 [ 1 31  1 33  1 35]]
```

Expected B16 - B18 Output:

```
[[False  True False  True False  True]
 [False  True False  True False  True]
 [False  True False  True False  True]
 [False  True False  True False  True]
 [False  True False  True False  True]
 [False  True False  True False  True]]
[ 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35]
[[ 1  1  1  3  1  5]
 [ 1  7  1  9  1 11]
 [ 1 13  1 15  1 17]
 [ 1 19  1 21  1 23]
 [ 1 25  1 27  1 29]
 [ 1 31  1 33  1 35]]
```

Goal: manipulate arrays

In [23]:
```
# B19 Given code
P = np.arange(36).reshape((6, 6))
```

Exercise B19. Set all the entries divisible by 3 to decreasing integers starting from 200. The first divisible-by-3 number should become 200, the second should become 199, and so on, with the final divisible-by-3 number becoming 189. Print the resulting array.

In [24]:
```
# B19: your code here
values = list(range(200, 188, -1))
P[P % 3 == 0] = values
print(P)
```

```
[[200   1   2 199   4   5]
 [198   7   8 197  10  11]
 [196  13  14 195  16  17]
 [194  19  20 193  22  23]
 [192  25  26 191  28  29]
 [190  31  32 189  34  35]]
```

Expected B19 Output:

```
[[200   1   2 199   4   5]
 [198   7   8 197  10  11]
 [196  13  14 195  16  17]
 [194  19  20 193  22  23]
 [192  25  26 191  28  29]
 [190  31  32 189  34  35]]
```

In [25]:
```
# B20 Given code
P = np.arange(36).reshape((6, 6))
```

Exercise B20. Add 10 to all of the even entries of the array. Print the resulting array.

In [26]:
```
# B20: your code here
P[P % 2 == 0] += 10
print(P)
```

```
[[10  1 12  3 14  5]
 [16  7 18  9 20 11]
 [22 13 24 15 26 17]
 [28 19 30 21 32 23]
 [34 25 36 27 38 29]
 [40 31 42 33 44 35]]
```

Expected B20 Output:

```
[[10  1 12  3 14  5]
 [16  7 18  9 20 11]
 [22 13 24 15 26 17]
 [28 19 30 21 32 23]
 [34 25 36 27 38 29]
 [40 31 42 33 44 35]]
```

Section 2: Pandas & SQL Skills

We'll be using a dataset on forest fires from *UCI* called *forest-fires.csv* (please use the version of the dataset in the HW zip file provided on Canvas). We will be using pandas to load ("read") the dataset. Run the below code that reads the csv file into a pandas dataframe named *df*. Remember that the csv file should live in the same directory as your local copy of this ipynb for this code to run.

In [27]:
```
df = pd.read_csv('forest-fires.csv')
```

C. DataFrames (10 points)

The main datatypes when using pandas are the *Series* and the *DataFrame*.

A Series is essentially a wrapper for a one-dimensional NumPy array. You can perform many operations on a Series just like you would with a NumPy array. The main difference is that it has an extra label/index for accessing each element, which we will learn about later on.

Goal: Create and operate on a pandas Series and DataFrame object.

A DataFrame is a two-dimensional table of data. A DataFrame has rows and columns, where each column is a Series. Columns can have different datatypes from each other.

In [28]:
```
# Exercises C1 - C3 Given code
l = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
s = pd.Series([0, 10, 20, 30, 40])
A = np.arange(12).reshape((6, 2))
```

Exercise C1. Convert list l to a pandas Series using the pd.Series() function. Print the Series. The numbers that appear in the left column are each row's label.

Exercise C2. Create a new Series object by multiplying the first Series s by 10. You can perform arithmetic directly on a Series just like we did when we used broadcasting on NumPy arrays. Print this new Series.

Exercise C3. Convert numpy array A to a pandas DataFrame with the `pd.DataFrame()` function. Use the `columns` argument to name the first column 'x' and the second column 'y'. Use the `index` argument to name the rows 'a', 'b', 'c', 'd', 'e', and 'f'. Print the resulting DataFrame.

In [29]:
```
# C1: your code here
l = pd.Series(l)
print(l)
```

```
# C2: your code here
s *= 10
print(s)

# C3: your code here
A = pd.DataFrame(A, columns=['x', 'y'], index = ['a', 'b', 'c', 'd', 'e', 'f
print(A)
```

```
0    a
1    b
2    c
3    d
4    e
5    f
6    g
dtype: object
0      0
1    100
2    200
3    300
4    400
dtype: int64
    x   y
a   0   1
b   2   3
c   4   5
d   6   7
e   8   9
f  10  11
```

Expected C1 - C3 Output:

```
0    a
1    b
2    c
3    d
4    e
5    f
6    g
dtype: object
0     0
1    100
2    200
3    300
4    400
dtype: int64
    x   y
a   0   1
b   2   3
c   4   5
d   6   7
e   8   9
f  10  11
```

Goal: Examine a dataframe.

Exercise C4. Use the DataFrame's `.head()` method to view the first ten rows of `df`, the new DataFrame you generated by reading in the `forest-fires.csv` file.

In [30]:
```
# C4: your code here
print(df.head(n=10))
```

```
   Temperature  Area
0         18.0  0.36
1         21.7  0.43
2         21.9  0.47
3         23.3  0.55
4         21.2  0.61
5         16.6  0.71
6         23.8  0.77
7         27.4  0.90
8         13.2  0.95
9         24.2  0.96
```

Expected C4 Output:

```
   Temperature  Area
0         18.0  0.36
1         21.7  0.43
2         21.9  0.47
3         23.3  0.55
4         21.2  0.61
5         16.6  0.71
6         23.8  0.77
7         27.4  0.90
8         13.2  0.95
9         24.2  0.96
```

D. Basic indexing (10 points)

Goal: Access rows using indices, labels and slicing. Understand when to use `.iloc` vs `.loc`.

Each row of a DataFrame has a label, which is like a special series that identifies each row of the DataFrame. If you don't specify a label when creating the DataFrame, the labels default to sequential integers. But DataFrames can have other labels as well, such as strings.

There are two main ways of accessing rows of a DataFrame:

- `df.iloc` accepts integer indices. For example, `df.iloc[10]` returns the row with index 10. You're familiar with this kind of indexing from NumPy arrays.
- `df.loc` accepts labels and boolean arrays. In subsequent exercises, we'll see how `df.loc` and `df.iloc` can yield different results. **We will most often be using `df.loc` in this class.**

In [31]:
```
# Exercises D1 - D2 Given code
cities_df = pd.DataFrame({'City': ['Ithaca', 'New York', 'Boston'], 'Populat
populations_df = pd.DataFrame({'Population': [32108, 8804190, 675647]}, inde
```

Exercise D1.

Part a: Use `DataFrame.iloc` indexing to print the row of `cities_df` with index 0.

Part b: Use `DataFrame.loc` indexing to print the row of `cities_df` with label 0. This will work because pandas defaults to integer labels when labels are not specified.

Exercise D2.

Part a: Use `DataFrame.iloc` indexing to print the row of `populations_df` with index 1.

Part b: Use `DataFrame.loc` indexing to print the row of `populations_df` with label 'New York'. Note that using `df.loc[1]`, as in the previous exercise, will produce a `KeyError` because `loc` searches for rows by their index value (here the city name) instead of an integer. This can be confusing because if you don't specify an index value, pandas defaults to setting the name of each row to its integer index.

Exercise D3. Use `df.iloc` to print the row of `df` (created in Exercise C4) with index 50.

Exercise D4. Use `df.iloc` and slicing to print the final 10 rows of `df`.

In [32]:
```
# D1 part a. your code here
print(cities_df.iloc[0])

# D1 part b. your code here
print(cities_df.loc[0])

# D2 part a. your code here
print(populations_df.iloc[1])

# D2 part b. your code here
print(populations_df.loc['New York'])

# D3 your code here
print(df.iloc[50])

# D4 your code here
print(df.iloc[-10:])
```

```
City          Ithaca
Population     32108
Name: 0, dtype: object
City          Ithaca
Population     32108
Name: 0, dtype: object
Population    8804190
Name: New York, dtype: int64
Population    8804190
Name: New York, dtype: int64
Temperature    13.3
Area            7.4
Name: 50, dtype: float64
     Temperature   Area
260         33.3  40.54
261         27.3  10.82
262         29.2   1.95
263         28.9  49.59
264         26.7   5.80
265         21.1   2.17
266         18.2   0.43
267         27.8   6.44
268         21.9  54.29
269         21.2  11.16
```

Expected D1 - D4 Output:

```
City          Ithaca
Population     32108
Name: 0, dtype: object
City          Ithaca
Population     32108
Name: 0, dtype: object
Population    8804190
Name: New York, dtype: int64
Population    8804190
Name: New York, dtype: int64
Temperature    13.3
Area            7.4
Name: 50, dtype: float64
     Temperature   Area
260         33.3  40.54
261         27.3  10.82
262         29.2   1.95
263         28.9  49.59
264         26.7   5.80
265         21.1   2.17
266         18.2   0.43
267         27.8   6.44
268         21.9  54.29
269         21.2  11.16
```

E. Advanced indexing: pandas (25 points)

Goal: Access an entire column of a DataFrame.

Exercise E1. Use indexing to create a new series variable that contains the `df` DataFrame's `Temperature` column. Print this series.

Exercise E2. Use indexing to create a new series variable that contains the `df` DataFrame's `Area` column. Print this series.

In [33]:
```
# E1: your code here
temp = pd.Series(df['Temperature'])
print(temp)

# E2: your code here
area = pd.Series(df['Area'])
print(area)
```

```
0      18.0
1      21.7
2      21.9
3      23.3
4      21.2
       ...
265    21.1
266    18.2
267    27.8
268    21.9
269    21.2
Name: Temperature, Length: 270, dtype: float64
0       0.36
1       0.43
2       0.47
3       0.55
4       0.61
       ...
265     2.17
266     0.43
267     6.44
268    54.29
269    11.16
Name: Area, Length: 270, dtype: float64
```

Expected E1 - E2 Output:

```
0      18.0
1      21.7
2      21.9
3      23.3
4      21.2
       ...
265    21.1
```

```
266    18.2
267    27.8
268    21.9
269    21.2
Name: Temperature, Length: 270, dtype: float64
0       0.36
1       0.43
2       0.47
3       0.55
4       0.61
       ...
265     2.17
266     0.43
267     6.44
268    54.29
269    11.16
Name: Area, Length: 270, dtype: float64
```

Goal: Check how many rows match a condition using `.sum()`.

Summing a boolean Series yields the number of entries in the Series that are `True`. You can catch a lot of errors when using boolean indexing if you always count the number of selected rows when doing any DataFrame manipulation.

Exercise E3. Use boolean indexing to select the rows corresponding to fires with temperatures greater than 15 degrees. Create a boolean Series where each entry corresponds to whether the fire with the given index has a temperature greater than 15 degrees. Print the number of `True` values in this series by using `.sum()`.

In [34]:
```
# E3: your code here
is_hot = temp > 15
print(is_hot.sum())
```

```
216
```

Expected E3 Output:

```
216
```

Goal: Select rows from a DataFrame based on values in the columns.

Exercise E4.

Part a: Create a boolean series named `hightemp_selector` where each entry corresponds to whether the fire with the given index has a temperature greater than 15

degrees. Print this series.

Part b: Use `df.loc` indexing to create a new DataFrame named `hightemp_df` that only contains the rows corresponding to these fires. You should call `hightemp_selector` in this line of code. Print the resulting smaller DataFrame.

Exercise E5.

Part a: Create a boolean series named `smallarea_selector` where each entry corresponds to whether the fire with the given index has an area less than 10. Print this series.

Part b: Use `df.loc` indexing to create a new DataFrame named `smallarea_df` that only contains the rows corresponding to these fires. You should call `smallarea_selector` in this line of code. Print the resulting smaller DataFrame.

In [35]:
```
# E4 part a. your code here
hightemp_selector = temp > 15
print(hightemp_selector)

# E4 part b. your code here
hightemp_df = df.loc[hightemp_selector]
print(hightemp_df)

# E5 part a. your code here
smallarea_selector = area < 10
print(smallarea_selector)

# E5 part b. your code here
smallarea_df = df.loc[smallarea_selector]
print(smallarea_df)
```

```
0      True
1      True
2      True
3      True
4      True
       ...
265    True
266    True
267    True
268    True
269    True
Name: Temperature, Length: 270, dtype: bool
     Temperature   Area
0           18.0   0.36
1           21.7   0.43
2           21.9   0.47
3           23.3   0.55
4           21.2   0.61
..           ...    ...
265         21.1   2.17
266         18.2   0.43
267         27.8   6.44
268         21.9  54.29
269         21.2  11.16

[216 rows x 2 columns]
0       True
1       True
2       True
3       True
4       True
       ...
265     True
266     True
267     True
268    False
269    False
Name: Area, Length: 270, dtype: bool
     Temperature  Area
0           18.0  0.36
1           21.7  0.43
2           21.9  0.47
3           23.3  0.55
4           21.2  0.61
..           ...   ...
262         29.2  1.95
264         26.7  5.80
265         21.1  2.17
266         18.2  0.43
267         27.8  6.44

[175 rows x 2 columns]
```

Expected E4 - E5 Output:

```
0      True
1      True
2      True
3      True
4      True
       ...
265    True
266    True
267    True
268    True
269    True
Name: Temperature, Length: 270, dtype: bool
     Temperature   Area
0           18.0   0.36
1           21.7   0.43
2           21.9   0.47
3           23.3   0.55
4           21.2   0.61
..           ...    ...
265         21.1   2.17
266         18.2   0.43
267         27.8   6.44
268         21.9  54.29
269         21.2  11.16

[216 rows x 2 columns]
0       True
1       True
2       True
3       True
4       True
       ...
265     True
266     True
267     True
268    False
269    False
Name: Area, Length: 270, dtype: bool
     Temperature  Area
0           18.0  0.36
1           21.7  0.43
2           21.9  0.47
3           23.3  0.55
4           21.2  0.61
..           ...   ...
262         29.2  1.95
264         26.7  5.80
265         21.1  2.17
266         18.2  0.43
267         27.8  6.44

[175 rows x 2 columns]
```

Exercise E6.

Using multiple conditions when indexing a DataFrame is very useful but syntactically confusing. We're going to have you do it in multiple steps with multiple variables first. Why multiple steps? Otherwise the single line of code looks like a magical formula. Once you get the individual steps, we'll have you combine the steps into one line of code that you can use later. A single line is also how you'll often see DataFrame indexing written. We want to show you how to recognize overall patterns that are getting combined for complex behavior.

Part a: Create a boolean series variable called `low_area_selector` where each entry corresponds to whether the fire with the given index has an area greater than or equal to 10. Create a second boolean series called `high_area_selector` where each entry corresponds to whether the fire with the given index has an area less than or equal to 20. Finally, create a third series called `selector` by taking the boolean *and* of the first two series (which can be done with the `&` operator). Print the `selector` series.

Part b: Use `df.loc` indexing to create a new DataFrame that only contains the rows corresponding to these fires. Print the resulting smaller DataFrame.

Part c: We've had you perform this indexing using multiple lines of code. Rewrite the previous part to use only one line of code. You will need to use parentheses around the boolean predicates, otherwise you will get an error. It's always good to be able to recognize error messages, so try this part without the parentheses first. Print the resulting dataframe. The output should be the same as in part b.

In [36]:
```
# E6 part a. your code here
low_area_selector = area >= 10
high_area_selector = area <= 20
selector = low_area_selector & high_area_selector
print(selector)

# E6 part b. your code here
limited_area = df.loc[selector]
print(limited_area)

# E6 part c. your code here
limited_area_one_liner = df.loc[(area >= 10) & (area <= 20)]
print(limited_area_one_liner)
```

```
0      False
1      False
2      False
3      False
4      False
       ...
265    False
266    False
267    False
268    False
269     True
Name: Area, Length: 270, dtype: bool
     Temperature   Area
56          24.6  10.01
57          23.5  10.02
58           5.8  10.93
59          21.5  11.06
60          13.9  11.24
61          22.6  11.32
62          21.6  11.53
63          12.4  12.10
64           8.8  13.05
65          20.2  13.70
66          15.1  13.99
67          22.1  14.57
68          22.9  15.45
69          20.7  17.20
70          19.6  19.23
101         15.4  10.13
111         18.9  10.34
127          5.1  11.19
129          4.6  17.85
130          4.6  10.73
145         24.8  14.29
156         21.5  15.64
157         17.1  11.22
173         20.8  13.06
183         16.8  12.64
184         13.8  11.06
185         10.3  18.30
189         16.2  16.33
192         21.3  12.18
193         20.9  16.00
220         20.9  15.34
243         26.4  10.08
254         26.4  16.40
259         32.3  14.68
261         27.3  10.82
269         21.2  11.16
     Temperature   Area
56          24.6  10.01
57          23.5  10.02
58           5.8  10.93
59          21.5  11.06
60          13.9  11.24
61          22.6  11.32
```

```
62          21.6  11.53
63          12.4  12.10
64           8.8  13.05
65          20.2  13.70
66          15.1  13.99
67          22.1  14.57
68          22.9  15.45
69          20.7  17.20
70          19.6  19.23
101         15.4  10.13
111         18.9  10.34
127          5.1  11.19
129          4.6  17.85
130          4.6  10.73
145         24.8  14.29
156         21.5  15.64
157         17.1  11.22
173         20.8  13.06
183         16.8  12.64
184         13.8  11.06
185         10.3  18.30
189         16.2  16.33
192         21.3  12.18
193         20.9  16.00
220         20.9  15.34
243         26.4  10.08
254         26.4  16.40
259         32.3  14.68
261         27.3  10.82
269         21.2  11.16
```

Expected E6 Output:

```
0      False
1      False
2      False
3      False
4      False
       ...
265    False
266    False
267    False
268    False
269     True
Name: Area, Length: 270, dtype: bool
     Temperature   Area
56          24.6  10.01
57          23.5  10.02
58           5.8  10.93
59          21.5  11.06
60          13.9  11.24
61          22.6  11.32
62          21.6  11.53
63          12.4  12.10
```

64	8.8	13.05
65	20.2	13.70
66	15.1	13.99
67	22.1	14.57
68	22.9	15.45
69	20.7	17.20
70	19.6	19.23
101	15.4	10.13
111	18.9	10.34
127	5.1	11.19
129	4.6	17.85
130	4.6	10.73
145	24.8	14.29
156	21.5	15.64
157	17.1	11.22
173	20.8	13.06
183	16.8	12.64
184	13.8	11.06
185	10.3	18.30
189	16.2	16.33
192	21.3	12.18
193	20.9	16.00
220	20.9	15.34
243	26.4	10.08
254	26.4	16.40
259	32.3	14.68
261	27.3	10.82
269	21.2	11.16

	Temperature	Area
56	24.6	10.01
57	23.5	10.02
58	5.8	10.93
59	21.5	11.06
60	13.9	11.24
61	22.6	11.32
62	21.6	11.53
63	12.4	12.10
64	8.8	13.05
65	20.2	13.70
66	15.1	13.99
67	22.1	14.57
68	22.9	15.45
69	20.7	17.20
70	19.6	19.23
101	15.4	10.13
111	18.9	10.34
127	5.1	11.19
129	4.6	17.85
130	4.6	10.73
145	24.8	14.29
156	21.5	15.64
157	17.1	11.22

```
173         20.8  13.06
183         16.8  12.64
184         13.8  11.06
185         10.3  18.30
189         16.2  16.33
192         21.3  12.18
193         20.9  16.00
220         20.9  15.34
243         26.4  10.08
254         26.4  16.40
259         32.3  14.68
261         27.3  10.82
269         21.2  11.16
```

Exercise E7.

Part a: Create a boolean series called `area_selector` where each entry corresponds to whether the fire with the given index has an area less than 15. Create a second boolean series called `temp_selector` where each entry corresponds to whether the fire with the given index has a temperature greater than 25. Finally, create a third series called `selector` by taking the boolean *and* of the first two series. Print the `selector` series.

Part b: Use `df.loc` indexing to create a new DataFrame that only contains the rows corresponding to these fires. Print the resulting smaller DataFrame:

Part c: As in the previous exercise, combine all of the above into one line. Remember to surround each predicate with parentheses. Print the final DataFrame. It should be the same as in part b.

In [37]:
```
# E7 part a. your code here
area_selector = area < 15
temp_selector = temp > 25
selector = area_selector & temp_selector
print(selector)

# E7 part b. your code here
filtered_data = df.loc[selector]
print(filtered_data)

# E7 part c. your code here
filtered_data_one_liner = df.loc[(area < 15) & (temp > 25)]
print(filtered_data_one_liner)
```

```
0      False
1      False
2      False
3      False
4      False
       ...
265    False
266    False
267     True
268    False
269    False
Length: 270, dtype: bool
     Temperature   Area
7           27.4   0.90
16          29.6   1.46
18          28.6   1.61
103         26.8   0.76
104         25.7   0.09
107         26.8   0.68
115         28.3   8.85
137         25.3   8.00
190         28.2   5.86
202         28.0   8.16
210         26.3   7.02
214         29.3   6.30
216         26.9   4.96
217         27.9   2.35
221         26.8   0.54
224         25.5   1.23
243         26.4  10.08
245         27.2   1.76
246         26.1   7.36
249         30.2   2.75
252         30.6   2.07
257         30.8   8.59
258         32.6   2.77
259         32.3  14.68
261         27.3  10.82
262         29.2   1.95
264         26.7   5.80
267         27.8   6.44
     Temperature   Area
7           27.4   0.90
16          29.6   1.46
18          28.6   1.61
103         26.8   0.76
104         25.7   0.09
107         26.8   0.68
115         28.3   8.85
137         25.3   8.00
190         28.2   5.86
202         28.0   8.16
210         26.3   7.02
214         29.3   6.30
216         26.9   4.96
217         27.9   2.35
```

```
221         26.8   0.54
224         25.5   1.23
243         26.4  10.08
245         27.2   1.76
246         26.1   7.36
249         30.2   2.75
252         30.6   2.07
257         30.8   8.59
258         32.6   2.77
259         32.3  14.68
261         27.3  10.82
262         29.2   1.95
264         26.7   5.80
267         27.8   6.44
```

Expected E7 Output:

```
0      False
1      False
2      False
3      False
4      False
       ...
265    False
266    False
267     True
268    False
269    False
Length: 270, dtype: bool
     Temperature   Area
7           27.4   0.90
16          29.6   1.46
18          28.6   1.61
103         26.8   0.76
104         25.7   0.09
107         26.8   0.68
115         28.3   8.85
137         25.3   8.00
190         28.2   5.86
202         28.0   8.16
210         26.3   7.02
214         29.3   6.30
216         26.9   4.96
217         27.9   2.35
221         26.8   0.54
224         25.5   1.23
243         26.4  10.08
245         27.2   1.76
246         26.1   7.36
249         30.2   2.75
252         30.6   2.07
257         30.8   8.59
258         32.6   2.77
```

```
259          32.3  14.68
261          27.3  10.82
262          29.2   1.95
264          26.7   5.80
267          27.8   6.44
     Temperature   Area
7            27.4   0.90
16           29.6   1.46
18           28.6   1.61
103          26.8   0.76
104          25.7   0.09
107          26.8   0.68
115          28.3   8.85
137          25.3   8.00
190          28.2   5.86
202          28.0   8.16
210          26.3   7.02
214          29.3   6.30
216          26.9   4.96
217          27.9   2.35
221          26.8   0.54
224          25.5   1.23
243          26.4  10.08
245          27.2   1.76
246          26.1   7.36
249          30.2   2.75
252          30.6   2.07
257          30.8   8.59
258          32.6   2.77
259          32.3  14.68
261          27.3  10.82
262          29.2   1.95
264          26.7   5.80
267          27.8   6.44
```

Goal: Summarize a Series with `.mean()` and `.sum()`.

Exercise E8. Print the mean temperature of all 270 fires in `df`.

Exercise E9. Print the mean area of all 270 fires in `df`.

Exercise E10. Print the total temperature of all 270 fires in `df`.

In [38]:
```
# E8: your code here
mean_temp = temp.mean()
print(mean_temp)

# E9: your code here
```

```
mean_area = area.mean()
print(mean_area)

# E10: your code here
total_area = temp.sum()
print(total_area)
```

```
19.31111111111111
24.600185185185182
5214.0
```

Expected E8 - E10 Output:

```
19.31111111111111
24.600185185185182
5214.0
```

Note: Depending on your machine, the last few digits of your output may not exactly match the output provided above.

F. Advanced indexing: SQL (25 points)

For this section, you will now answer similar problems as in section E, but using SQL instead of pandas. You do not need the pandas indexing to write your SQL code. You will be using the duckdb library. Remember that you must wrap your SQL query in quotes inside the parentheses of `duckdb.sql()`. Finally, in order to convert this output into a pandas dataframe, you must use `.df()` at the end of your code. See the documentation *here*.

If you want to split lines of SQL code within parentheses (so `SELECT` is on a different line from `FROM`, and so on), you can use the backslash character `\`.

Goal: Access an entire column of a DataFrame.

Exercise F1. Use DuckDB and SQL to create a new variable that contains the `df` DataFrame's `Temperature` column. Print this variable.

Exercise F2. Use DuckDB and SQL to create a new variable that contains the `df` DataFrame's `Area` column. Print this variable.

In [39]:
```
# F1: your code here
temp_sql = duckdb.sql("SELECT Temperature FROM df")
print(temp_sql.df())

# F2: your code here
area_sql = duckdb.sql("SELECT Area FROM df")
print(area_sql.df())
```

```
     Temperature
0           18.0
1           21.7
2           21.9
3           23.3
4           21.2
..           ...
265         21.1
266         18.2
267         27.8
268         21.9
269         21.2

[270 rows x 1 columns]
      Area
0     0.36
1     0.43
2     0.47
3     0.55
4     0.61
..     ...
265   2.17
266   0.43
267   6.44
268  54.29
269  11.16

[270 rows x 1 columns]
```

Expected F1 - F2 Output:

```
     Temperature
0           18.0
1           21.7
2           21.9
3           23.3
4           21.2
..           ...
265         21.1
266         18.2
267         27.8
268         21.9
269         21.2

[270 rows x 1 columns]
      Area
0     0.36
1     0.43
2     0.47
3     0.55
4     0.61
..     ...
265   2.17
266   0.43
```

```
267   6.44
268  54.29
269  11.16

[270 rows x 1 columns]
```

Goal: Check how many rows match a condition using `WHERE` and `COUNT`.

Exercise F3.

Part a: Use `WHERE` to create a new DataFrame including only fires where the temperature is less than 6 degrees. Print this new DataFrame.

Part b: Use `COUNT` to return the number of rows in the DataFrame created in part a. Print this number.

In [40]:
```
# F3 part a: your code here
temp_sql = duckdb.sql("SELECT * FROM df WHERE Temperature < 6")
print(temp_sql.df())


# F3 part b: your code here
temp_sql_count = duckdb.sql("SELECT COUNT(*) FROM df WHERE Temperature < 6")
print(temp_sql_count.df())
```

```
    Temperature   Area
0           5.3   2.14
1           5.8   4.61
2           5.8  10.93
3           5.1  26.00
4           4.8   8.98
5           5.1  11.19
6           5.1   5.38
7           4.6  17.85
8           4.6  10.73
9           4.6  22.03
10          4.6   9.77
11          2.2   9.27
12          5.1  24.77
13          4.6   5.39
14          5.1   2.14
15          4.6   6.84
   count_star()
0            16
```

Expected F3 Output:

```
    Temperature   Area
0           5.3   2.14
1           5.8   4.61
```

```
2           5.8  10.93
3           5.1  26.00
4           4.8   8.98
5           5.1  11.19
6           5.1   5.38
7           4.6  17.85
8           4.6  10.73
9           4.6  22.03
10          4.6   9.77
11          2.2   9.27
12          5.1  24.77
13          4.6   5.39
14          5.1   2.14
15          4.6   6.84

   count_star()
0            16
```

Goal: Select rows from a DataFrame based on values in the columns.

Exercise F4. Select the rows of the original DataFrame corresponding to fires with an area less than 10. Print the resulting smaller DataFrame.

Exercise F5. Select the rows corresponding to fires with an area between 10 and 20 (inclusive). As we did with pandas, you will select rows based on multiple conditions. Use `AND` to select on two conditions in a single SQL statement. Print the resulting DataFrame.

Exercise F6. Select the rows corresponding to fires that have *both* a temperature less than 15 and an area greater than 25. Print the resulting DataFrame.

In [41]:
```
# F4: your code here
bounded_fire_sql = duckdb.sql("SELECT * FROM df WHERE Area < 10")
print(bounded_fire_sql.df())


# F5: your code here
range_area_sql = duckdb.sql("SELECT * FROM df WHERE Area > 10 AND Area < 20"
print(range_area_sql.df())

# F6: your code here
two_conditions_sql = duckdb.sql("SELECT * FROM df WHERE Temperature < 15 AND
print(two_conditions_sql.df())
```

```
     Temperature  Area
0           18.0  0.36
1           21.7  0.43
2           21.9  0.47
3           23.3  0.55
4           21.2  0.61
..           ...   ...
170         29.2  1.95
171         26.7  5.80
172         21.1  2.17
173         18.2  0.43
174         27.8  6.44

[175 rows x 2 columns]
    Temperature   Area
0          24.6  10.01
1          23.5  10.02
2           5.8  10.93
3          21.5  11.06
4          13.9  11.24
5          22.6  11.32
6          21.6  11.53
7          12.4  12.10
8           8.8  13.05
9          20.2  13.70
10         15.1  13.99
11         22.1  14.57
12         22.9  15.45
13         20.7  17.20
14         19.6  19.23
15         15.4  10.13
16         18.9  10.34
17          5.1  11.19
18          4.6  17.85
19          4.6  10.73
20         24.8  14.29
21         21.5  15.64
22         17.1  11.22
23         20.8  13.06
24         16.8  12.64
25         13.8  11.06
26         10.3  18.30
27         16.2  16.33
28         21.3  12.18
29         20.9  16.00
30         20.9  15.34
31         26.4  10.08
32         26.4  16.40
33         32.3  14.68
34         27.3  10.82
35         21.2  11.16
   Temperature   Area
0          5.1  26.00
1         11.0  27.35
2         12.4  30.32
3         11.0  36.85
```

```
4         13.4  37.02
5         10.1  51.78
6         13.7  61.13
```

Expected F4 - F6 Output:

```
     Temperature  Area
0           18.0  0.36
1           21.7  0.43
2           21.9  0.47
3           23.3  0.55
4           21.2  0.61
..           ...   ...
170         29.2  1.95
171         26.7  5.80
172         21.1  2.17
173         18.2  0.43
174         27.8  6.44

[175 rows x 2 columns]
    Temperature   Area
0          24.6  10.01
1          23.5  10.02
2           5.8  10.93
3          21.5  11.06
4          13.9  11.24
5          22.6  11.32
6          21.6  11.53
7          12.4  12.10
8           8.8  13.05
9          20.2  13.70
10         15.1  13.99
11         22.1  14.57
12         22.9  15.45
13         20.7  17.20
14         19.6  19.23
15         15.4  10.13
16         18.9  10.34
17          5.1  11.19
18          4.6  17.85
19          4.6  10.73
20         24.8  14.29
21         21.5  15.64
22         17.1  11.22
23         20.8  13.06
24         16.8  12.64
25         13.8  11.06
26         10.3  18.30
27         16.2  16.33
28         21.3  12.18
29         20.9  16.00
30         20.9  15.34
```

```
31          26.4  10.08
32          26.4  16.40
33          32.3  14.68
34          27.3  10.82
35          21.2  11.16
   Temperature   Area
0          5.1  26.00
1         11.0  27.35
2         12.4  30.32
3         11.0  36.85
4         13.4  37.02
5         10.1  51.78
6         13.7  61.13
```

Goal: Summarize a Series with `Avg()` and `Sum()`.

Exercise F7. Use SQL commands to print the mean temperature of all 270 fires in `df`.

Exercise F8. Use SQL commands to print the mean area of all 270 fires in `df`.

Exercise F9. Use SQL commands to print the total area of all 270 fires in `df`.

In [42]:
```
# F7: your code here
temp_avg_sql = duckdb.sql("SELECT AVG(Temperature) FROM df")
print(temp_avg_sql.df())

# F8: your code here
area_avg_sql = duckdb.sql("SELECT AVG(Area) FROM df")
print(area_avg_sql.df())

# F9: your code here
area_sum_sql = duckdb.sql("SELECT SUM(Area) FROM df")
print(area_sum_sql.df())
```

```
  avg(Temperature)
0        19.311111
   avg(Area)
0  24.600185
   sum(Area)
0    6642.05
```

Expeceted F7 - F9 Output:

```
   avg(Temperature)
0        19.311111
   avg(Area)
0  24.600185
   sum(Area)
0    6642.05
```

How long it did it take you to complete this homework?

Add your answer (an estimate in hours) in the cell below.

Time taken here: 3-4

ChatGPT usage: essentially just used informationally to get function documentation.

Conversation Link: https://chatgpt.com/share/68ba3674-dcf4-8004-96fd-2a2e98239a12