CARTE-Enbridge Bootcamp

Lab 1-1a

In this lab, we will introduce you to the very basics of programming using Python. We'll cover:

- Variables and Data Types
- Basic Operations
- Lists
- Loops and Conditionals

Don't worry if you're new to programming; we'll walk you through each step. Notebooks

We're using notebooks for this lab. Write your code in cells like this one. To run a cell, press the play button above or hit ctrl+enter. To write text like this, set the cell to "Markdown" from the dropdown menu. Getting Started

First, let's make sure Python is working. **>> Run the code in the next cell** to print a welcome message.

In [1]:
```python
print("Welcome to Python!")
```

```
Welcome to Python!
```

Variables and Data Types

In Python, we store information in variables. Think of a variable as a box where you can put stuff like numbers or text.

>> Run the code in the next cell to create a variable and print it.

In [2]:
```python
my_variable = 10
print("My variable contains:", my_variable)
```

```
My variable contains: 10
```

Python has different types of data:

- Integer (whole numbers)
- Float (decimal numbers)
- String (text)

>> Run the code in the next cell to see examples.

In [3]:
```python
integer_example = 5
float_example = 5.0
string_example = "Hello"

print("Integer:", integer_example)
print("Float:", float_example)
print("String:", string_example)
```

```
Integer: 5
Float: 5.0
String: Hello
```

Basic Operations

You can perform basic math operations in Python.

>> Run the code in the next cell to see some examples.

In [4]:
```python
addition = 5 + 5
subtraction = 10 - 5
multiplication = 2 * 5
division = 10 / 2

print("Addition:", addition)
print("Subtraction:", subtraction)
print("Multiplication:", multiplication)
print("Division:", division)
```

```
Addition: 10
Subtraction: 5
Multiplication: 10
Division: 5.0
```

Notice that the result of division is a float, even if the result is a whole number. This is because the result could be a decimal number. If you want to perform integer division, use the `//` operator instead:

In [5]:
```python
integer_division = 5 // 2
print("Integer Division:", integer_division)
```

```
Integer Division: 2
```

Lists

A list is a collection of items. Lists are useful for storing multiple items in a single variable.

>> Run the code in the next cell to create and print a list.

In [6]:
```python
my_list = [1, 2, 3, 4, 5]
print("My list contains:", my_list)
```

```
My list contains: [1, 2, 3, 4, 5]
```

You can access items in a list by their position. In Python (and many other programming languages), the first item in a list is at position 0.

In [7]:
```
print("The first item in my list is:", my_list[0])
print("The third item in my list is:", my_list[2])
```

```
The first item in my list is: 1
The third item in my list is: 3
```

Loops and Conditionals

Loops and conditionals are used to control the flow of a program. They allow you to run certain code only if certain conditions are met.

>> Run the code in the next cell to see a simple loop and conditional.

In [8]:
```
for number in my_list:  # Loop through each item in the list
    if number % 2 == 0:  # Check if the number is divisible by 2
        print(number, "is even.")
    else:
        print(number, "is odd.")
```

```
1 is odd.
2 is even.
3 is odd.
4 is even.
5 is odd.
```

Your Turn

Now it's your turn to write some code. In the next cell, write a program that does the following:

1. Create a list of numbers from 1 to 15
2. Loop through the list
3. If the number is divisible by 3, print "Fizz"
4. If the number is divisible by 5, print "Buzz"
5. If the number is divisible by both 3 and 5, print "FizzBuzz"
6. Otherwise, print the number itself

>> Write your code in the next cell. You can copy and paste the code from the previous cell and modify it.

In [9]:
```
# Answer

my_list = list(range(1, 16))  # Create a list of numbers from 1 to 15
for number in my_list:  # Loop through each item in the list
    if number % 3 == 0 and number % 5 == 0:  # Check if the number is divisi
```

```
        print("FizzBuzz")
    elif number % 3 == 0:  # Check if the number is divisible by 3
        print("Fizz")
    elif number % 5 == 0:  # Check if the number is divisible by 5
        print("Buzz")
    else:
        print(number)
```

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
```

Functions

Functions are a way to organize your code. They allow you to reuse code without having to write it again.

>> Run the code in the next cell to define a simple function, then use it.

In [10]:
```
def greet(name):
    print("Hello,", name)
```

In [11]:
```
greet("John")
greet("Jane")
```

```
Hello, John
Hello, Jane
```

Libraries

Libraries are collections of functions and methods that allow you to perform actions without writing the code yourself. Let's import the `math` library and try some functions.

>> Run the code in the next cell to import the `math` library and use some functions.

In [12]:
```
import math
```

```
print("Pi is:", math.pi)
print("The square root of 16 is:", math.sqrt(16))
```

```
Pi is: 3.141592653589793
The square root of 16 is: 4.0
```

NumPy

When we work with lots of data in Python, we use a library called NumPy. NumPy allows us to work with arrays, which are like lists but with more functionality. The code underlying NumPy isn't written in Python, but in C, which is much faster and allows us to work with large amounts of data. Many other libraries in Python are built on top of NumPy.

>> Run the code in the next cell to import NumPy and create an array.

In [13]:
```
import numpy as np  # This is the standard way to import NumPy - we are just
```

In [14]:
```
my_array = np.array(
    [1, 2, 3, 4, 5]
)  # We can create an array from a standard Python list
print("Array:", my_array)
```

```
Array: [1 2 3 4 5]
```

To get a sense of how much faster NumPy is than standard Python, let's create a large array and time how long it takes to sum all the elements when using a standard Python list vs. a NumPy array.

In [15]:
```
my_large_list = list(range(1000000))  # Create a list of 1 million numbers
my_large_array = np.array(my_large_list)  # Create a NumPy array from the li
```

In [16]:
```
import time  # Import the time library to time our code
```

In [17]:
```
start_time = time.time()  # Get the current time
sum(my_large_list)  # Sum all the elements in the list
print(
    "Time to sum a list:", time.time() - start_time, "seconds"
)  # Print the time elapsed
```

```
Time to sum a list: 0.005071878433227539 seconds
```

In [18]:
```
start_time = time.time()  # Get the current time
np.sum(my_large_array)  # Sum all the elements in the array
print(
    "Time to sum an array:", time.time() - start_time, "seconds"
)  # Print the time elapsed
```

```
Time to sum an array: 0.0005605220794677734 seconds
```

Both of these times might seem fast, but remember that we're only summing 1 million numbers. In data science, we often work with millions or billions of data points, so that fact that NumPy is about 10x faster than standard Python is a big deal.

Arrays can be multi-dimensional. We can create a 2D array like this:

In [19]:
```
my_2d_array = np.array([[1, 2, 3], [4, 5, 6]])
print("2D Array:\n", my_2d_array)  # The \n character creates a new line
```

```
2D Array:
 [[1 2 3]
 [4 5 6]]
```

We can access the shape, number of dimensions, data type, and number of elements in our array as follows:

In [20]:
```
print("Shape:", my_2d_array.shape)
print("Number of Dimensions:", my_2d_array.ndim)
print("Data Type:", my_2d_array.dtype.name)
print("Number of Elements:", my_2d_array.size)
```

```
Shape: (2, 3)
Number of Dimensions: 2
Data Type: int64
Number of Elements: 6
```

Indexing

Python lets us access elements in a list or array using a very convenient set of rules called indexing. We can access the first element in a list or array like before:

In [21]:
```
print("First element:", my_array[0])
```

```
First element: 1
```

But what if we want to access the last element in the list? We could count how many elements are in the list, then subtract 1, but that's a lot of work when we are dealing with huge amounts of data. Instead, we can use negative indexing:

In [22]:
```
print("Last element:", my_array[-1])
print("Second to last element:", my_array[-2])
```

```
Last element: 5
Second to last element: 4
```

We can also access a range of elements using the `:` operator. The syntax is `[start:end]`, where `start` is the index of the first element we want to access and `end` is the index of the first element we don't want to access. Note that the element at index `end` is not included - I know this can be a bit confusing at first!

In [23]:
```
print("First three elements:", my_array[0:3])  # Gives us elements 0, 1 and
```

```
First three elements: [1 2 3]
```

If we leave out the `start` or `end` index, Python will assume we want to start at the beginning or go all the way to the end, respectively:

```
In [24]: print("First three elements:", my_array[:3])  # Think of it as "up to 3rd el
         print("Last three elements:", my_array[-3:])  # Elements 3, 4 and 5
```

```
First three elements: [1 2 3]
Last three elements: [3 4 5]
```

This is a lot to get used to, but we will be using indexing a lot in this course, so you will get plenty of practice.

Optional Exercise: Advanced Weather Analysis

In this optional exercise, you'll analyze four weeks of weather data for a hypothetical city. The data includes daily high temperatures (in Celsius) and weather conditions.

Data

Here is four weeks' worth of weather data:

```
temperatures = [
    [15, 18, 12, 20, 16, 19, 17],  # Week 1
    [22, 21, 23, 19, 17, 18, 20],  # Week 2
    [25, 27, 26, 24, 22, 23, 24],  # Week 3
    [15, 14, 12, 11, 13, 14, 15]   # Week 4
]

conditions = [
    ["Rain", "Sunny", "Cloudy", "Sunny", "Rain", "Cloudy",
"Sunny"],  # Week 1
    ["Sunny", "Sunny", "Cloudy", "Rain", "Rain", "Sunny", "Sunny"],
# Week 2
    ["Sunny", "Sunny", "Sunny", "Cloudy", "Cloudy", "Sunny",
"Sunny"], # Week 3
    ["Snow", "Snow", "Snow", "Cloudy", "Sunny", "Snow", "Snow"]
# Week 4
]

days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday", "Sunday"]
```

Tasks

1. Calculate the average temperature for each week. Store the results in a list called `average_temperatures`.

2. Create a function that finds the day with the highest temperature for each week and returns those days.
3. Create a function that finds the day with the lowest temperature for each week and returns those days.
4. Create a function that counts the number of "Sunny" days in each week.
5. Use NumPy to find the overall average temperature for the four weeks.
6. Use NumPy to find the highest and lowest temperatures across all four weeks.

In [25]:
```
temperatures = [
    [15, 18, 12, 20, 16, 19, 17],  # Week 1
    [22, 21, 23, 19, 17, 18, 20],  # Week 2
    [25, 27, 26, 24, 22, 23, 24],  # Week 3
    [15, 14, 12, 11, 13, 14, 15]   # Week 4
]

conditions = [
    ["Rain", "Sunny", "Cloudy", "Sunny", "Rain", "Cloudy", "Sunny"],  # Week
    ["Sunny", "Sunny", "Cloudy", "Rain", "Rain", "Sunny", "Sunny"],    # Wee
    ["Sunny", "Sunny", "Sunny", "Cloudy", "Cloudy", "Sunny", "Sunny"], # Wee
    ["Snow", "Snow", "Snow", "Cloudy", "Sunny", "Snow", "Snow"]        # Wee
]

days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday",
```

In [26]:
```
# 1. Calculate the average temperature for each week. Store the results in a

average_temperatures = []
for week in temperatures:
    average_temperatures.append(np.mean(week))
print(average_temperatures)
```

```
[16.714285714285715, 20.0, 24.428571428571427, 13.428571428571429]
```

In [27]:
```
# 2. Create a function that finds the day with the highest temperature for e

def highest_temp_day(weeks):
    highest_temp_days = []
    for week in weeks:
        highest_temp_days.append(days[np.argmax(week)])
    return highest_temp_days

print(highest_temp_day(temperatures))
```

```
['Thursday', 'Wednesday', 'Tuesday', 'Monday']
```

In [28]:
```
# 3. Create a function that finds the day with the lowest temperature for ea

def lowest_temp_day(weeks):
    lowest_temp_days = []
    for week in weeks:
        lowest_temp_days.append(days[np.argmin(week)])
    return lowest_temp_days
```

```
print(lowest_temp_day(temperatures))
```

```
['Wednesday', 'Friday', 'Friday', 'Thursday']
```

In [29]:
```
# 4. Create a function that counts the number of "Sunny" days in each week.

def sunny_days(weeks):
    sunny_days = []
    for week in weeks:
        sunny_days.append(np.count_nonzero(np.array(week) == "Sunny"))
    return sunny_days

print(sunny_days(conditions))
```

```
[3, 4, 5, 1]
```

In [30]:
```
# 5. Use NumPy to find the overall average temperature for the four weeks.

print(np.mean(temperatures))
```

```
18.642857142857142
```

In [31]:
```
# 6. Use NumPy to find the highest and lowest temperatures across all four w

print(np.max(temperatures))
print(np.min(temperatures))
```

```
27
11
```