ILP 2024 : Computing

W3S2
29 May 2024

Objective

- Error Handling
- Plotting
- Numpy library

Testing

What is Program Testing?

- It involves executing a program with the intent of finding errors or bugs.
- Program testing is the process of verifying that a software application performs its intended functions correctly.

– Edsger W. Dijkstra

Why is Program Testing Important?

- Ensures software quality and reliability.
- Identifies defects early in the development cycle.
- Validates that the software meets requirements.

Program testing can be used to show the presence of bugs, but never to show their absence!”

Testing types

- **Unit Testing**:
  - Tests individual components or units of code in isolation.
  - Uses frameworks like `unittest` or `pytest` in Python.
- **Integration Testing**:
  - Tests the interaction between integrated components or modules.
  - Ensures that units work together as expected.
- **System Testing**:
  - Tests the entire system as a whole.
  - Validates that all components work together in a real-world environment.
- **Regression Testing**:
  - Re-runs previous tests to ensure that changes to the codebase haven't introduced new bugs.

Testing example

```
def function(my_list):
    # Remove max and min from list
    min_val = min(my_list)
    max_val = max(my_list)
    while(min_val in my_list):
        my_list.remove(min_val)
    while(max_val in my_list):
        my_list.remove(max_val)
    return my_list
```

```
# Test case 1: a normal, good looking list,
# with a single max value and a single min value
my_list = [1,2,3,4,5]
print(function(my_list))
```

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


Remove the min and max from the given list

Some more test cases

```
# Test case 2: a normal, good looking list,
# with multiple occurences of the max and min values
my_list = [1,1,2,3,4,5,5,5]
print(function(my_list))
```

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

```
# Test case 3: a list with only min and max values
my_list = [1,1,5,5,5]
print(function(my_list))
```

```
[]
```

```
# Test case 4: a list with only min and max values,
# and the min/max values are identical
my_list = [5,5,5]
print(function(my_list))
```

```
[]
```

```
# Test case 5: an empty list
my_list = []
print(function(my_list))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[6], line 3
      1 # Test case 5: an empty list
      2 my_list = []
----> 3 print(function(my_list))

Cell In[1], line 3, in function(my_list)
      1 def function(my_list):
      2     # Remove max and min from list
----> 3     min_val = min(my_list)
      4     max_val = max(my_list)
      5     while(min_val in my_list):

ValueError: min() iterable argument is empty
```

Testing

```python
def function_v2(my_list):

    # Warning: Need to cover for empty list case
    if(len(my_list) == 0):
        return []
    else:
        # Remove max and min from list
        min_val = min(my_list)
        max_val = max(my_list)
        while(min_val in my_list):
            my_list.remove(min_val)
        while(max_val in my_list):
            my_list.remove(max_val)
        return my_list
```

```python
# Test case 5: an empty list
my_list = []
print(function_v2(my_list))
```

```
[]
```

Update our function


```python
# Test case 6: a non-empty list
# with non-numerical values
my_list = ["Hello", "What", "is", "up?"]
# Function works without errors,
# but is it really the expected behavior?
print(function_v2(my_list))
```

```
['What', 'is']
```

Is it really the expected behavior?

Testing

Testing can sometimes feel like a game of cats and dogs because:

- Some bugs may remain hidden until triggered by specific conditions.
- Ultimately, testing is an ongoing process, where developers continuously refine and improve their tests to ensure the reliability and quality of their code.

Practise Test

Write a recursive function to calculate the factorial of a given number.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))
```

```
120
```

Try these test cases :)

1. Test Case : `factorial(-5)`
2. Test Case : `factorial(3.5)`
3. Test Case : `factorial(1000)`
4. Test Case : `factorial(1)`
5. Test Case : `factorial(100000)`

Assertion

- **assert** statements are used for debugging code in Python.

assert [condition], [error message]

- An **error message** [AssertionError] is displayed when the **condition** is **False**.

```
bool1 = True
message = "Nothing will be displayed."
assert bool1, message
```

```
bool1 = False
message = "Assertion test failed, interrupted program, error message here."
assert bool1, message
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[26], line 3
      1 bool1 = False
      2 message = "Assertion test failed, interrupted program, error message here."
----> 3 assert bool1, message

AssertionError: Assertion test failed, interrupted program, error message here.
```

```
def divide(x, y):
    assert y != 0, "Divisor cannot be zero"
    return x / y

result = divide(10, 1)
print(result)
```

```
10.0
```

```
result = divide(10, 0)
print(result)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[25], line 1
----> 1 result = divide(10, 0)
      2 print(result)

Cell In[24], line 2, in divide(x, y)
      1 def divide(x, y):
----> 2     assert y != 0, "Divisor cannot be zero"
      3     return x / y

AssertionError: Divisor cannot be zero
```

`__debug__` constant

- **The** `__debug__` is a built-in constant in Python that is set to `True` by default

```
def divide(x, y):
    #assert y != 0, "Divisor cannot be zero"
    if __debug__:
        if y == 0:
            raise AssertionError("Divisor cannot be zero")
    return x / y

result = divide(10, 0)
print(result)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[68], line 8
      5             raise AssertionError("Divisor cannot be zero")
      6     return x / y
----> 8 result = divide(10, 0)
      9 print(result)

Cell In[68], line 5, in divide(x, y)
      3 if __debug__:
      4     if y == 0:
----> 5         raise AssertionError("Divisor cannot be zero")
      6 return x / y

AssertionError: Divisor cannot be zero
```

Asserting on types

- Another interesting function is the `isinstance()` one, which is used for type checking.
- It receives two arguments.
- The first one is a variable/object,
- The second a type (int, float, str, list, etc.)
- It returns **True**, if the variable is of said type and **False** otherwise


```
x = 10
# Returns True, because x is an int type
print(isinstance(x, int))
# Returns False, because x is an int type
print(isinstance(x, str))
```

```
True
False
```

```
y = 15
print(isinstance(x, y))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[31], line 2
      1 y = 15
----> 2 print(isinstance(x, y))

TypeError: isinstance() arg 2 must be a type, a tuple of types, or a union
```

Benefits of Using Assertions

Facilitates Debugging: Assertions help identify and diagnose errors by catching invalid states or unexpected conditions early in development.

Improves Code Reliability: By checking assumptions about program state, assertions improve the overall reliability and correctness of code.

Practise test

1. Implement a function that calculates the average of a list of numbers. Use assertions to ensure that the input list is not empty before performing the calculation.

2. Create a Python script that prompts the user to enter their age. Use assertions to verify that the entered age is within a valid range (e.g., between 0 and 120).

3. Create a program that simulates a simple ATM machine. Use assertions to verify that the user's input for the withdrawal amount is less than or equal to the available balance.

4. Develop a program that simulates a basic calculator. Use assertions to ensure that the user's input for the arithmetic operation is one of the supported operations (e.g., addition, subtraction, multiplication, division).

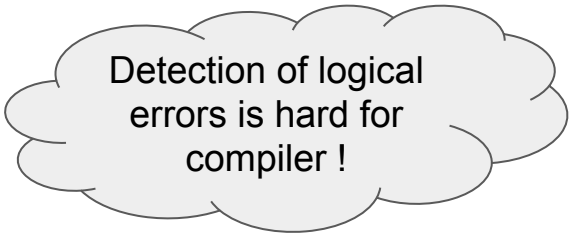
Error messages

- Error messages are exceptions, which were caught by the Python compiler program, when it attempted to execute your code.
- Typically, assertions, which did not pass, so that the program could execute normally.

```
# Add 1 to all numbers in list_numbers
list_numbers = ["0", "1", "2", "3"]
add_1_list = []
for number in list_numbers:
    val = number + 1
    add_1_list.append(str(val))
print(add_1_list)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
Cell In[32], line 5
      3 add_1_list = []
      4 for number in list_numbers:
----> 5     val = number + 1
      6     add_1_list.append(str(val))
      7 print(add_1_list)

TypeError: can only concatenate str (not "int") to str
```


- They usually consist of an approximate location of where the error occurred,
- And a standardized error message, attempting to explain the type of error encountered.



Debugging

- **Debugging** is the art/process of **detecting and removing** of existing and potential **errors** (also called as 'bugs') in a code that can cause it to behave unexpectedly or crash.
- Python interpreter tries to provide a location of where the error occurred in its error message.
    - Unfortunately, it can't detect all the errors.
- Developer (you) need to pinpoint the location of the error, with some prints, to control which parts of the program work fine and which do not.


```
# Add 1 to all numbers in list_numbers
list_numbers = ["0", "1", "2", "3"]
add_1_list = []
print("Ok")
for number in list_numbers:
    print("Ok1")
    val = number + 1
    print("Ok2")
    add_1_list.append(str(val))
    print("Ok3")
print(add_1_list)
```

```
Ok
Ok1
---------------------------------------------------------------
TypeError                                 Traceback (mo
Cell In[33], line 7
      5 for number in list_numbers:
      6     print("Ok1")
----> 7     val = number + 1
      8     print("Ok2")
```

Python Try Except

- The `try` block lets you test a block of code for errors.
- The `except` block lets you handle the error.

```
#Try_Except
x = 10
try:
  print(x)
except:
  print("An exception occurred")
```

```
10
```



```
del(x)
try:
  print(x)
except:
  print("An exception occurred")
```

```
An exception occurred
```

Contd …

- The **`else`** block lets you execute code when there is no error.
- The **`finally`** block lets you execute code, regardless of the result of the try- and except blocks.


```
x = 10
try:
  print(x)
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
finally:
  print("The 'try except' is finished")
```

```
10
Nothing went wrong
The 'try except' is finished
```

- Nested **`try-except`** block

```
try:
  try:
    print(xyz)
  except:
    print("Inner - Something went wrong")
except:
  print("Outer - Something went wrong")
```

```
Inner - Something went wrong
```

Plotting

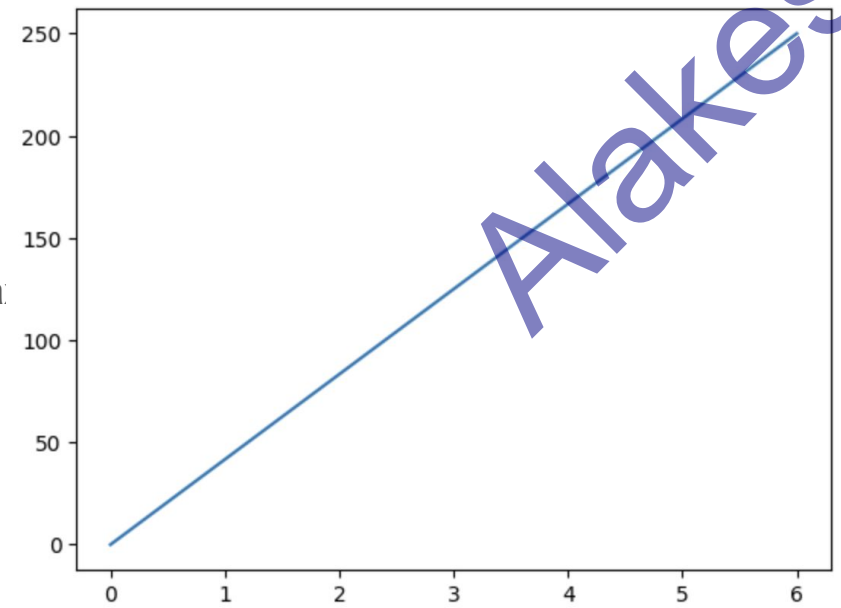
Matplotlib

- **Matplotlib** is a low level graph plotting library in python that offers data visualization.
- Matplotlib was created by John D. Hunter.
- Matplotlib is open source and we can use it freely.
- For installation
  - `pip install matplotlib`


```
Last login: Wed Mar 27 12:12:30 on ttys000
alakesh_kalita@ECCLT2203807s-MacBook-Pro ~ % pip install matplotlib
Collecting matplotlib
  Downloading matplotlib-3.8.3-cp312-cp312-macosx_11_0_arm64.whl.metadata (5.8 k
B)
Collecting contourpy>=1.0.1 (from matplotlib)
  Downloading contourpy-1.2.0-cp312-cp312-macosx_11_0_arm64.whl.metadata (5.8 kB
)
Collecting cycler>=0.10 (from matplotlib)
  Downloading cycler-0.12.1-py3-none-any.whl.metadata (3.8 kB)
Collecting fonttools>=4.22.0 (from matplotlib)
  Downloading fonttools-4.50.0-cp312-cp312-macosx_10_9_universal2.whl.metadata (
159 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 159.4/159.4 kB 6.0 MB/s eta 0:00:00
Collecting kiwisolver>=1.3.1 (from matplotlib)
```

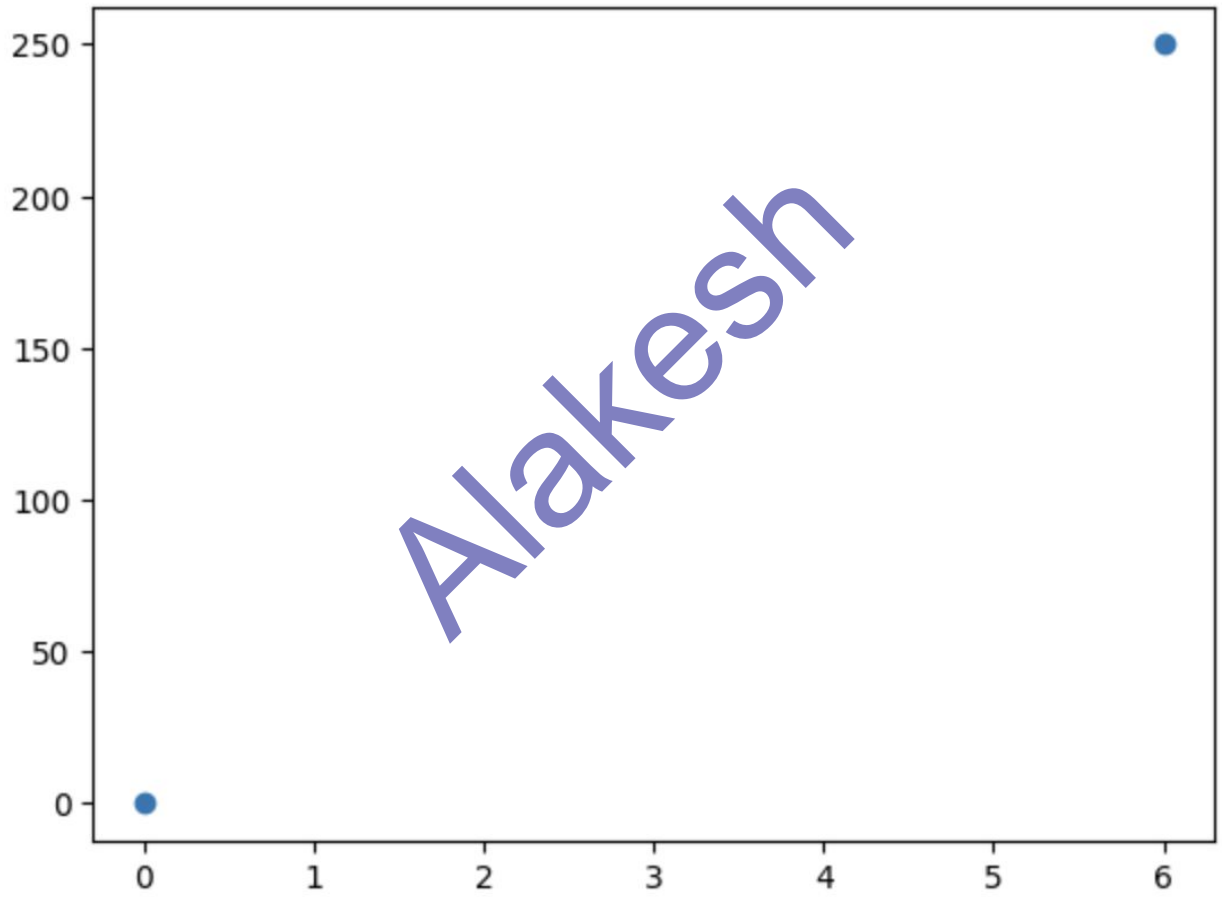
T

```
import matplotlib.pyplot as plt

xpoints = [0, 6]
ypoints = [0, 250]

plt.plot(xpoints, ypoints)
plt.show()
```



X-a

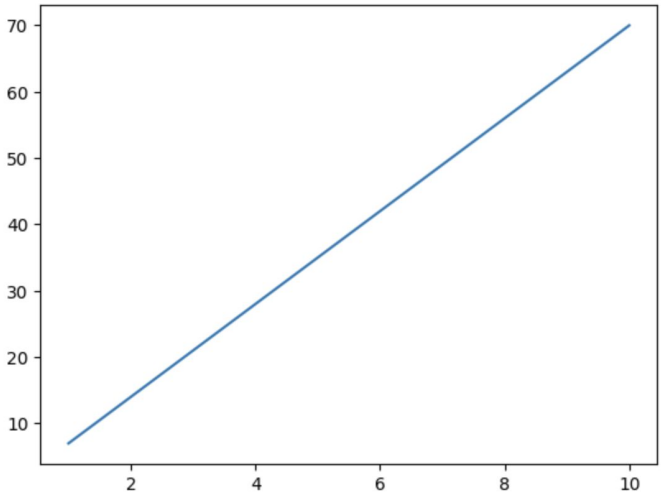
```
plt.plot(xpoints, ypoints, 'o')
plt.show()
```



Practise test

Can you quickly plot the multiplication table of 7

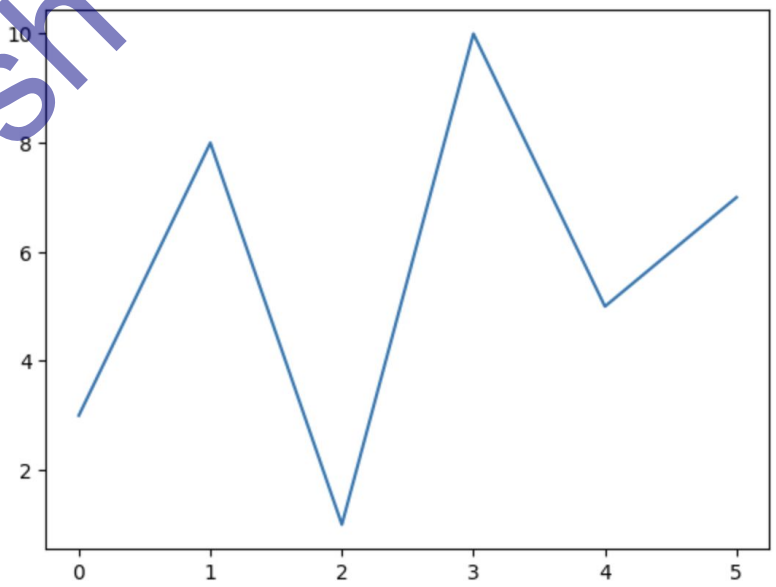

```
plt.plot(xpoints, ypoints)
plt.show()
```



Contd …

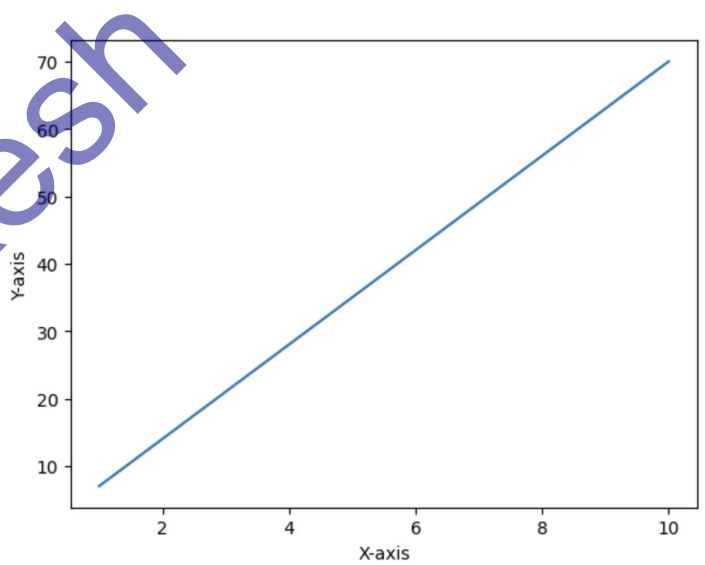
- If you don’t mention either the values of X-axis or Y-axis, that will take the default values from 0 to number of items in Provided-axis

```
ypoints = [3, 8, 1, 10, 5, 7]

plt.plot(ypoints)
plt.show()
```



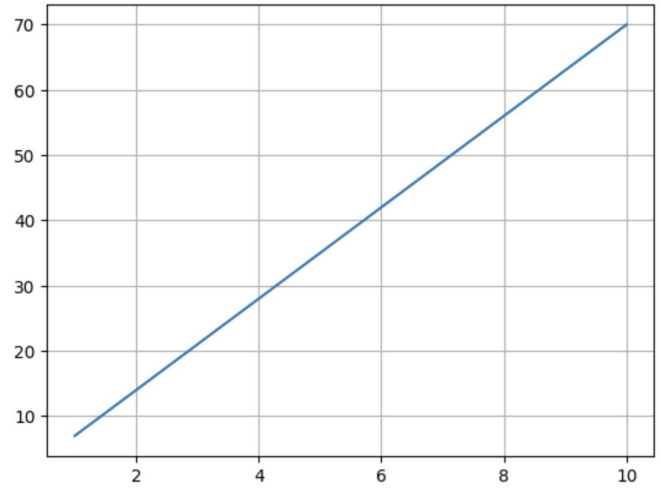
Adding labels

```python
import matplotlib.pyplot as plt

# Generate data
xpoints = [x for x in range(1, 11)]
ypoints = [x * 7 for x in range(1, 11)]

# Plot data
plt.plot(xpoints, ypoints)

# Add labels
plt.xlabel('X-axis')
plt.ylabel('Y-axis')

# Display plot
plt.show()
```

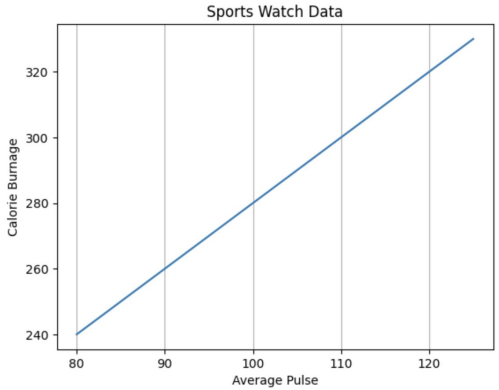



Contd …

```
plt.plot(xpoints, ypoints)
plt.grid()
plt.show()
```

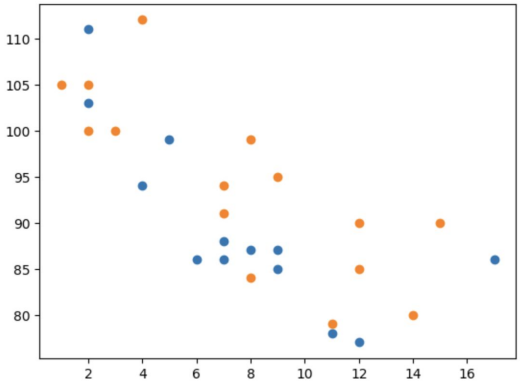



```
import numpy as np
import matplotlib.pyplot as plt

x = [80, 85, 90, 95, 100, 105, 110, 115, 120, 125]
y = [240, 250, 260, 270, 280, 290, 300, 310, 320, 330]

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)

plt.grid(axis = 'x')

plt.show()
```

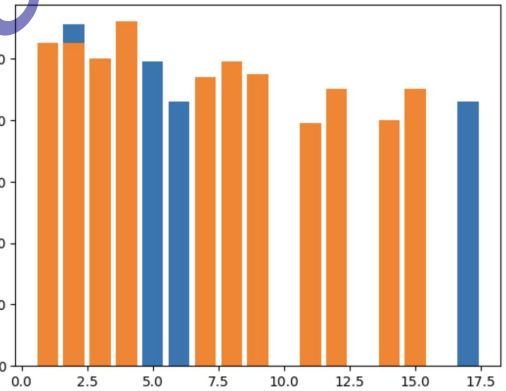


Contd …

```python
import matplotlib.pyplot as plt

#day one, the age and speed of 13 cars:
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
plt.scatter(x, y)

#day two, the age and speed of 15 cars:
x = [2,2,8,1,15,8,12,9,7,3,11,4,7,14,12]
y = [100,105,84,105,90,99,90,95,94,100,79,112,91,80,85]
plt.scatter(x, y)

plt.show()
```

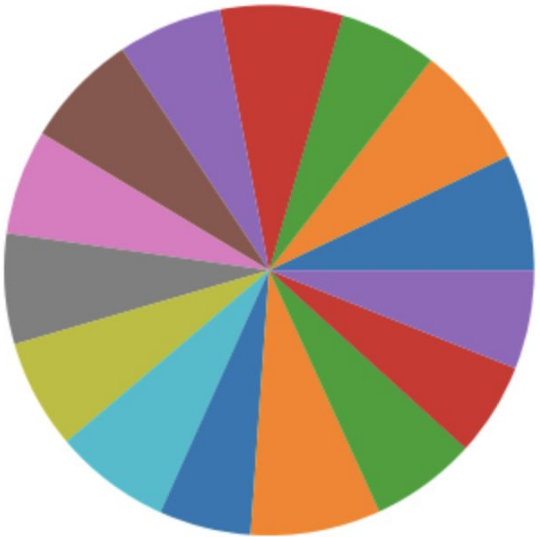



```python
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
plt.bar(x, y)

#day two, the age and speed of 15 cars:
x = [2,2,8,1,15,8,12,9,7,3,11,4,7,14,12]
y = [100,105,84,105,90,99,90,95,94,100,79,112,91,80,85]
plt.bar(x, y)

plt.show()
```



Contd …

```
plt.pie(y)
plt.show()
```



```
y = [35, 25, 25, 15]
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels)
plt.show()
```