Collection: *Lists*

Python Programming Fundamental

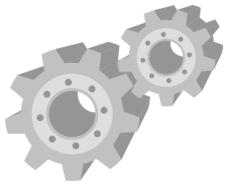
Husni

Department of Informatics Engineering
University of Trunojoyo Madura

Outline

- Processing collection of data using lists
- List creation and manipulation
- Various operations on lists

Storing Collection of Data



- Python provides many built-in data types to store a group of data
  - `list` – an ordered collection of objects
  - `tuple` – immutable version of `list`
  - `dict` – a collection of key-value mapping
  - `set` – an unordered collection of distinct objects
- And a lot more in the standard `collections` module
- This course will focus only on `list`

Quick Task: Find Average

- Find the average score of students.

24

28

26

32

```
Enter student score (or ENTER to finish): 24
Enter student score (or ENTER to finish): 26
Enter student score (or ENTER to finish): 28
Enter student score (or ENTER to finish): 32
Enter student score (or ENTER to finish):
Average score is 27.5
```

Find Average – Solution

- This should be straightforward

```
sum = 0
count = 0
while True:
    ans = input("Enter student score (or ENTER to finish): ")
    if ans == "":
        break
    score = float(ans)
    sum = sum + score
    count = count + 1

avg = sum/count
print(f"Average score is {avg}")
```

Task: Find Below Average

- Similar to Find Average, but also list the scores that are below the average

24

28

26

32

```
Enter student score (or ENTER to finish): 24
Enter student score (or ENTER to finish): 26
Enter student score (or ENTER to finish): 28
Enter student score (or ENTER to finish): 32
Enter student score (or ENTER to finish):
Average score is 27.5
Scores below average:
24
26
```

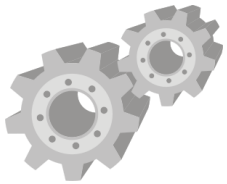
Find Below Average – Ideas

- We need to keep track of every single score
- Declaring one variable for one score is very inflexible

```
s1 = float(input("Enter student score: "))
s2 = float(input("Enter student score: "))
s3 = float(input("Enter student score: "))
  :
```

We cannot even control how many times to read scores

Storing a list of data



- Python provides the `list` data type to store a list of objects

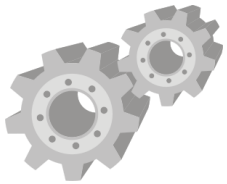
```
scores = []
while True:
    score = input("Enter score (or ENTER to finish): ")
    if score == "":
        break
    score = int(score)
    scores.append(score)

print("Scores are:", scores)
```

Create an empty list

Append a new element

```
Enter score (or ENTER to finish): 24
Enter score (or ENTER to finish): 26
Enter score (or ENTER to finish): 28
Enter score (or ENTER to finish): 32
Enter score (or ENTER to finish):
Scores are: [24, 26, 28, 32]
```

List Creation



- Create an empty list

```
list1 = []
```

- Create a list containing 4 integers: 20, 12, 8, 6

```
list2 = [20, 12, 8, 6]
```

- Create a list containing 3 floats: 1.2, 3.1, 8.0

```
list3 = [1.2, 3.1, 8.0]
```

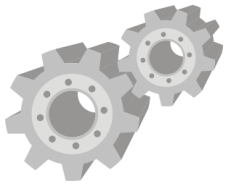
- Create a list containing 2 strings: "Hello", "Goodbye"

```
list4 = ["Hello", "Goodbye"]
```

- Create a list with mixed data types

```
list5 = ["Hello", 9, 3.8]
```

List Member Access

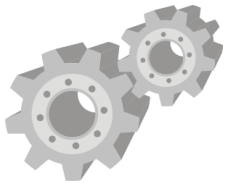


- Members in a list can be accessed using the [ ] operator with an index (similar to strings)

```
>>> lst = [8,3,2,5,3,1,6]
>>> lst[0]
8
>>> lst[1]
3
>>> lst[-1]
6
```

- **Reminder:** index starts from 0

Lists Are Mutable



- Unlike strings, list's contents can be changed

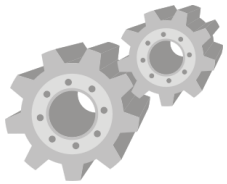
```
>>> lst = [8,3,9,5,3,1,6]
>>> lst
[8, 3, 9, 5, 3, 1, 6]
>>> lst[2] = 38
>>> lst
[8, 3, 38, 5, 3, 1, 6]
```

- A new element can be added using the `list.append()` method (a *method* is a function bound to an object)

```
>>> lst
[8, 3, 38, 5, 3, 1, 6]
>>> lst.append(72)
>>> lst
[8, 3, 38, 5, 3, 1, 6, 72]
```

PythonTutor

List's Length and List Traversal



- The function `len()` returns the length of a list
- A list can be used as a sequence of a **`for`** loop

```
>>> lst = [8,3,2,5,3,1,6]
>>> len(lst)
7
>>> for x in lst:
...     print(x)
8
3
2
5
3
1
6
```

Task Revisited: Find Below Average

- Let us get back to the task

24

28

26

32

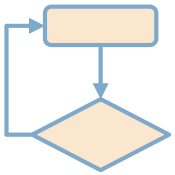
```
Enter student score (or ENTER to finish): 24
Enter student score (or ENTER to finish): 26
Enter student score (or ENTER to finish): 28
Enter student score (or ENTER to finish): 32
Enter student score (or ENTER to finish):
Average score is 27.5
Scores below average:
24
26
```

Find Below Average – Ideas

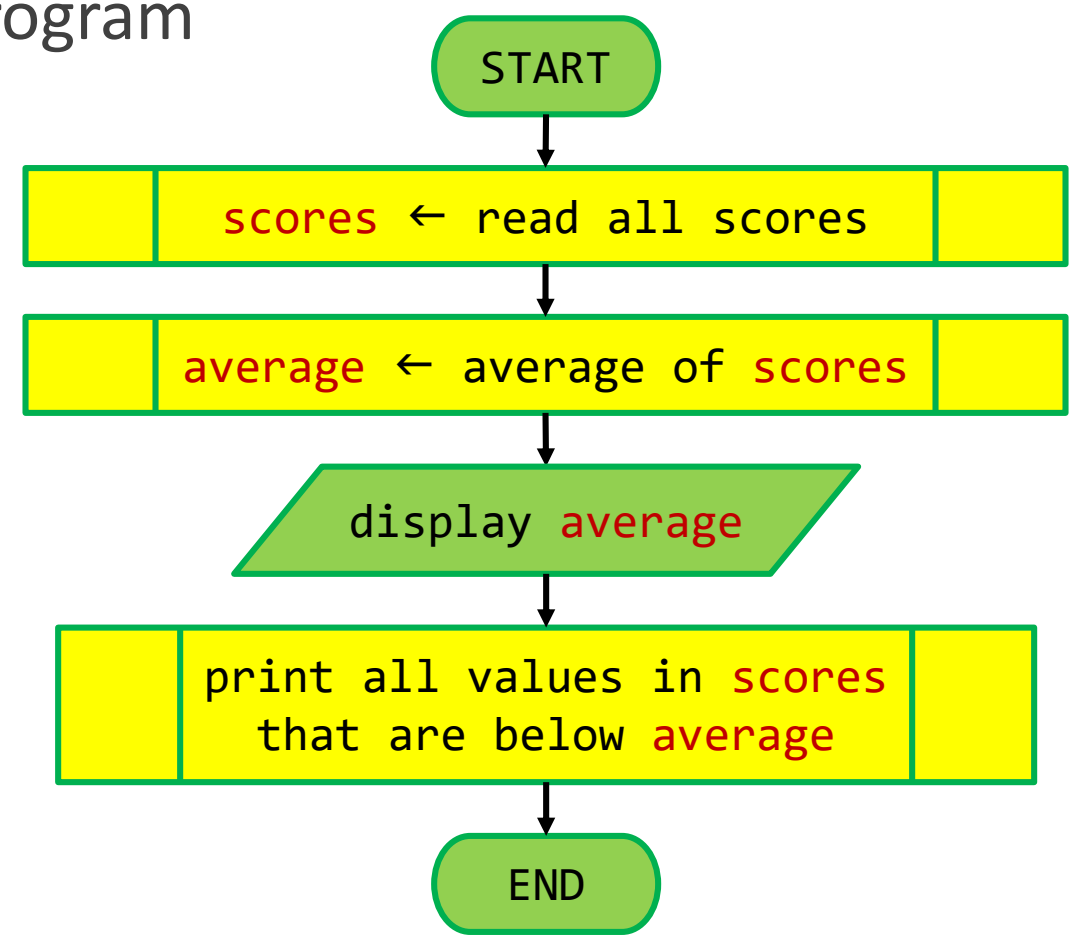
- We will divide the task into smaller subtasks
  - `read_scores()` – reads and returns scores as a list
  - `compute_average(scores)` – computes the average from a list of scores
  - `print_below(scores,value)` – prints only scores that are below the given value

- We will then write a subroutine for each of these subtasks

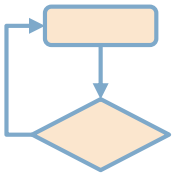
Find Below Average – Steps



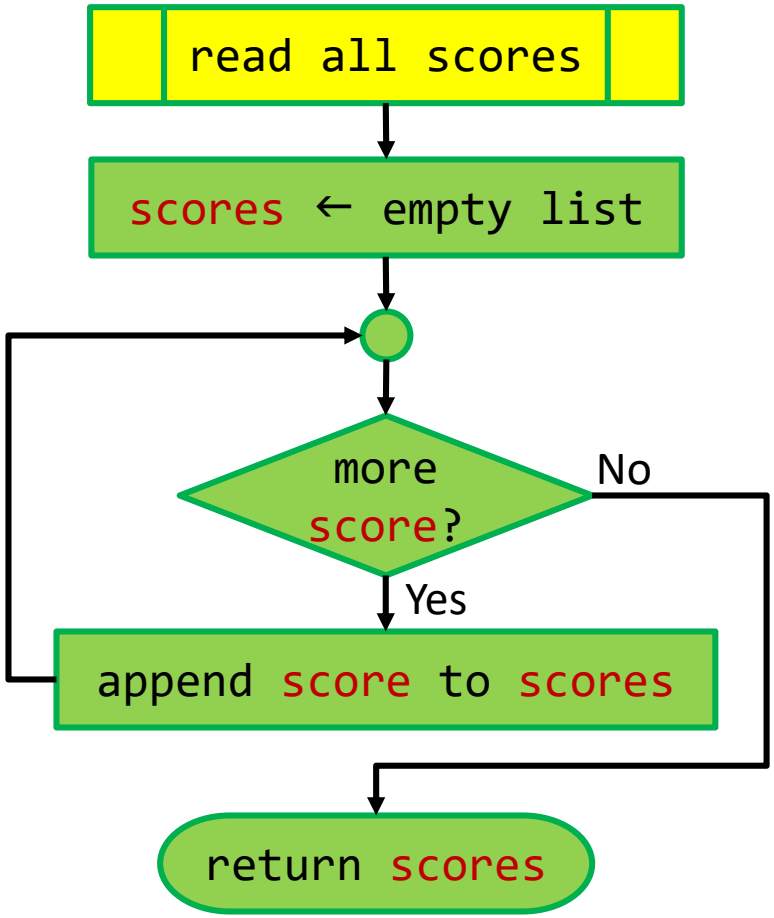
- Main program



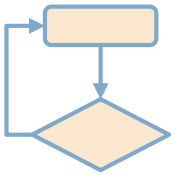
Find Below Average – Steps



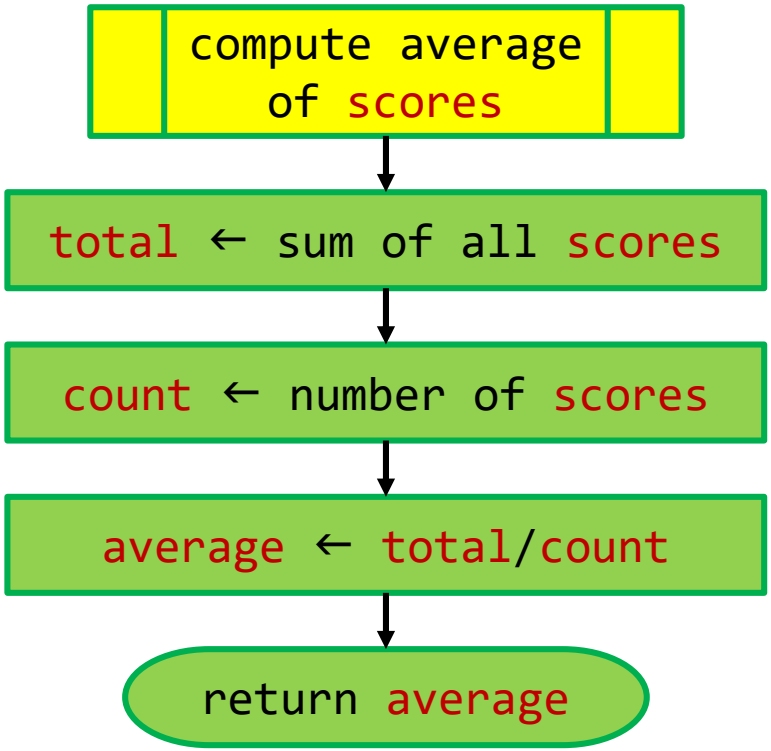
- `read_scores()` subroutine



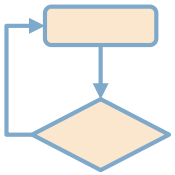
Find Below Average – Steps



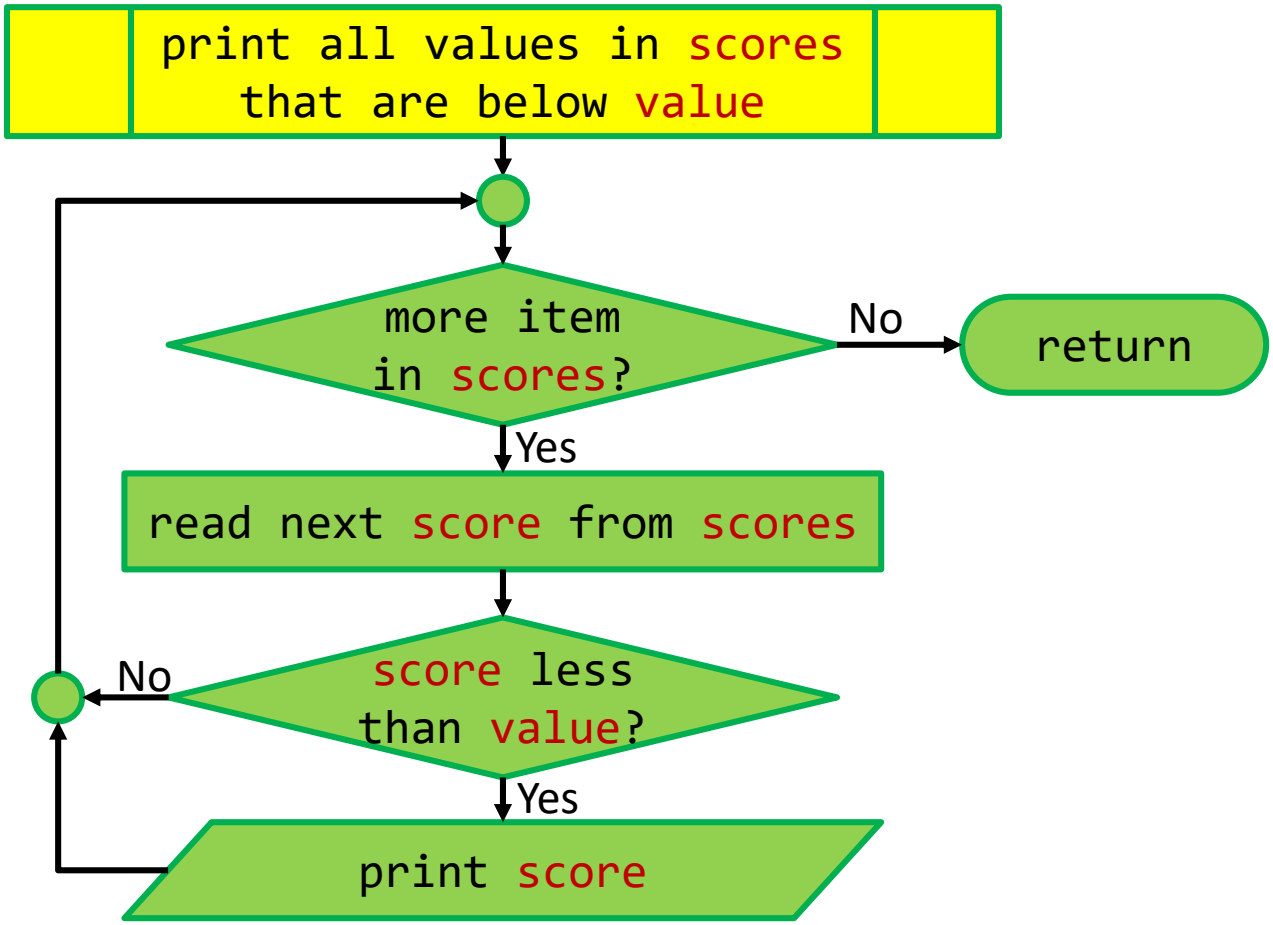
- `compute_average(scores)` subroutine



Find Below Average – Steps



- `print_below(scores,value)` subroutine



Find Below Average – Subroutines

read all scores

```
def read_scores():
    scores = []
    while True:
        ans = input("Enter student score (or ENTER to finish): ")
        if ans == "":
            break
        scores.append(int(ans))
    return scores
```

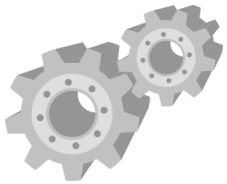
compute average of scores

```
def compute_average(scores):
    sum = 0
    for s in scores:
        sum = sum + s
    return sum/len(scores)
```

print all values in scores that are below value

```
def print_below(scores,value):
    for s in scores:
        if s < value:
            print(s)
```

Built-in Function: `sum()`



- `sum(lst)` returns the summation of all the items in the list `lst`

```
>>> sum([1,2,3,4])
10
>>> sum([10,50,21,27])
108
>>> sum(range(101))
5050
```

- Therefore, `compute_average()` can be rewritten as

```
def compute_average(scores):
    sum = 0
    for s in scores:
        sum = sum + s
    return sum/len(scores)
```

```
def compute_average(scores):
    return sum(scores)/len(scores)
```

Find Below Average – Testing

- Once we have defined all subroutines, let us test them one by one
- Testing `read_scores()`

```
def read_scores():
    scores = []
    while True:
        ans = input("Enter student score...")
        if ans == "":
            break
        scores.append(int(ans))
    return scores
```

```
>>> scores = read_scores()
Enter student score (or ENTER to finish): 28
Enter student score (or ENTER to finish): 26
Enter student score (or ENTER to finish): 32
Enter student score (or ENTER to finish): 37
Enter student score (or ENTER to finish):
>>> scores
[28.0, 26.0, 32.0, 37.0]
```

Find Below Average – Testing

- Testing `compute_average()`

```
def compute_average(scores):
    return sum(scores)/len(scores)
```

```
>>> compute_average([1])
1.0
>>> compute_average([1,2])
1.5
>>> compute_average([1,2,3])
2.0
>>> compute_average([1.2,4.6,5.1])
3.633333333333333
```

Find Below Average – Testing

- Testing `print_below()`

```
>>> print_below([6,2,4,8,1,2],3)
2
1
2
>>> print_below([6,2,4,8,1,2],4.5)
2
4
1
2
>>> print_below([6,2,4,8,1,2],6)
2
4
1
2
```

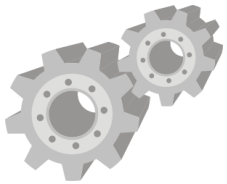
```
def print_below(scores,value):
    for s in scores:
        if s < value:
            print(s)
```

Find Below Average – Main

- Once we have tested all subroutines, let us write the main program

```
scores = read_scores()
avg = compute_average(scores)
print(f"Average score is {avg}")
print("Scores below average:")
print_below(scores,avg)
```

Finding Min and Max



- In addition to `sum()`, Python also provides `min()` and `max()` functions
  - `min(lst)` returns the minimum value in the list `lst`
  - `max(lst)` returns the maximum value in the list `lst`

```
>>> nums = [6,2,4,8,1,2]
>>> min(nums)
1
>>> max(nums)
8
```

Task: Score Statistics

- Read a list of scores and report the summary table, along with average, minimum, and maximum scores

24

28

26

32

```
Enter student score (or ENTER to finish): 24
Enter student score (or ENTER to finish): 26
Enter student score (or ENTER to finish): 28
Enter student score (or ENTER to finish): 32
Enter student score (or ENTER to finish):
Student #1 score: 24
Student #2 score: 26
Student #3 score: 28
Student #4 score: 32
Average score is 27.5
Minimum score is 24
Maximum score is 32
```

Score Statistics – Ideas

- Most subroutines from the previous example can be reused (`read_scores`, `compute_average`)
- Min and max can be computed using the built-in functions
- The only challenge is the summary table part

```
scores = read_scores()
show_score_summary(scores)
avg_score = compute_average(scores)
min_score = min(scores)
max_score = max(scores)
print(f"Average score is {avg_score}")
print(f"Minimum score is {min_score}")
print(f"Maximum score is {max_score}")
```

Score Statistics – Ideas

- The summary needs to display the order of each student's score

```
Enter student score (or ENTER to finish): 24
Enter student score (or ENTER to finish): 26
Enter student score (or ENTER to finish): 28
Enter student score (or ENTER to finish): 32
Enter student score (or ENTER to finish):
Student #1 score: 24
Student #2 score: 26
Student #3 score: 28
Student #4 score: 32
Average score is 27.5
Minimum score is 24
Maximum score is 32
```

- A `for` loop with a combination of `len()` and `range()` can help

Score Statistics – Program

- Only the `show_score_summary()` function is shown here

```
def show_score_summary(scores):
    for i in range(len(scores)):
        print(f"Student #{i+1} score: {scores[i]}")
```

- Let's test it

```
>>> show_score_summary([31,56,73,48])
Student #1 score: 31
Student #2 score: 56
Student #3 score: 73
Student #4 score: 48
```

List vs. String

- Lists and strings share many similarity
  - Member access with `[]`
  - The `len()` function
  - Their use with `for` loop
- The main difference is **lists are mutable but strings are immutable**

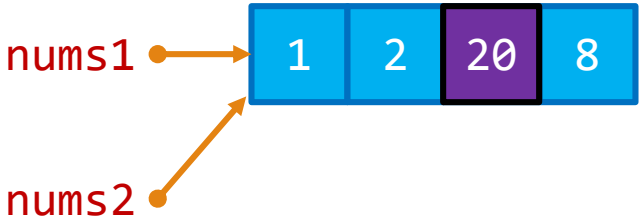
```
>>> L = [1,2,3,4,5]
>>> L[3] = 8
>>> L
[1, 2, 3, 8, 5]
```

```
>>> s = "Hello"
>>> s[3] = "c"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Caveats – Lists are mutable

- Assigning two or more names to the same list may have undesired effect

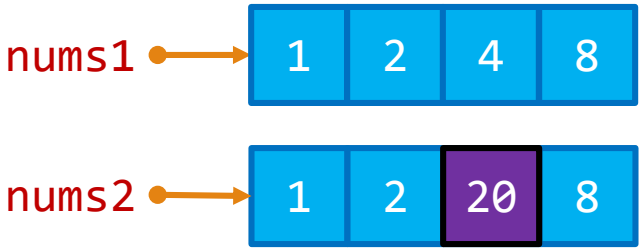
```
>>> nums1 = [1,2,4,8]
>>> nums2 = nums1
>>> nums2[2] = 20
>>> nums1
[1, 2, 20, 8]
```



PythonTutor

- To make a copy of a list, use `list()` function instead

```
>>> nums1 = [1,2,4,8]
>>> nums2 = list(nums1)
>>> nums2[2] = 20
>>> nums1
[1, 2, 4, 8]
>>> nums2
[1, 2, 20, 8]
```



PythonTutor

Bonus – *Membership Test*

- Using the `in` operator

```
>>> numbers = [5,1,8,2,7]
>>> 5 in numbers
True
>>> 9 in numbers
False
```

This is a Boolean expression

- The `in` operator also works with strings

```
>>> s = "Hello"
>>> "e" in s
True
>>> "L" in s
False
>>> "lo" in s
True
```

Membership Test – Example

- The following code counts the number of vowels (a,e,i,o,u) in the given text

```
text = input("Enter a text: ")
count = 0
for c in text:
    if c in "AEIOUaeiou":
        count = count + 1
print(f"Found {count} vowel(s)")
```

```
Enter a text: Hello
Found 2 vowel(s)
```

```
Enter a text: Good morning
Found 4 vowel(s)
```

Bonus – *List Slicing*

- Slicing creates a new list as a subset of an existing list
- Slicing syntax for a list L:

```
L(start:stop:step)
```

- The newly created list is:

```
[L[start],L[start+step],L[start+2step],…]
```

  - The last member DOES NOT include L[*stop*]
  - *start* can be omitted, implying 0
  - *stop* can be omitted, implying list's length
  - *step* can be omitted, implying 1

Examples – List Slicing

```
>>> L = [1,4,9,16,25,36,49]
>>> L[2:4]
[9, 16]
>>> L[1:]
[4, 9, 16, 25, 36, 49]
>>> L[:5]
[1, 4, 9, 16, 25]
>>> L[1:6:2]
[4, 16, 36]
>>> L[::-1]
[49, 36, 25, 16, 9, 4, 1]
>>> L[:]
[1, 4, 9, 16, 25, 36, 49]
```

Specifying start and stop

Specifying only start

Specifying only stop

Specifying start, stop, and step

Specifying a negative step

Specifying nothing (copying list)

Example – List Slicing

- The following code slices a list of month names into four quarters

```
months = [
    'Jan','Feb','Mar','Apr','May','Jun',
    'Jul','Aug','Sep','Oct','Nov','Dec'
]

q1 = months[0:3]
q2 = months[3:6]
q3 = months[6:9]
q4 = months[9:12]

print("Quarter 1:", q1)
print("Quarter 2:", q2)
print("Quarter 3:", q3)
print("Quarter 4:", q4)
```

```
Quarter 1: ['Jan', 'Feb', 'Mar']
Quarter 2: ['Apr', 'May', 'Jun']
Quarter 3: ['Jul', 'Aug', 'Sep']
Quarter 4: ['Oct', 'Nov', 'Dec']
```

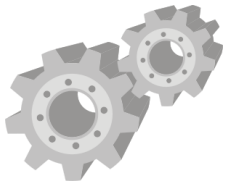
Conclusion

- A list is used to store ordered collection of values as one single object
- List members can be added and changed at any time
- A `for` loop can be used to iterate over each member
- `len()`, `sum()`, `min()`, and `max()` are some built-in functions that work with lists
- Lists are quite similar to strings, except that lists are mutable but strings are immutable

References

- Python data structures:
  - https://docs.python.org/3/tutorial/datastructures.html
- Common sequence operations
  - https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range

Syntax Summary (1)



- Creating a list

```
L = [member0,member1,...]
```

- Accessing the member at i^{th} position (starting at 0)

```
L[i]
```

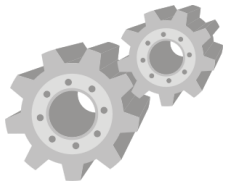
- Appending a new member at the end of the list

```
L.append(new_member)
```

- Finding the list's length

```
len(L)
```

Syntax Summary (2)



- Finding the sum, minimum, and maximum of all members in the list (numerical members only)

```
sum(L)
```

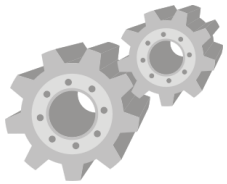
```
min(L)
```

```
max(L)
```

- Traversing list's members

```
for member in L:
    ...
```

Syntax Summary (bonus)



- Checking whether *value* is in the list

```
value in L
```

- Create a slicing of the list

```
L[start:stop:step]
```

  - *start*, *stop*, and *step* are all optional