Python Lecture 5: Data Structures

September 18, 2025

1 Review of Functions from Lecture 4

To build on the concepts from Lecture 4, we start with a quick refresh of built-in functions, importing packages, and discovering library functions. This connects to our new topic by showing how functions can manipulate data structures.

1.1 Refreshing Built-in Functions

Built-in functions like `print()`, `len()`, and `input()` are always available. Here's a simple program to refresh their use, incorporating user input and output as seen in Lecture 4 examples.

```
def refresh_builtins():
    name = input("Enter your name: ")
    print("Hello, " + name + "! Your name has " + str(len(
    name)) + " characters.")

refresh_builtins()
```

Example Input: Alice
Output: Hello, Alice! Your name has 5 characters.

This function takes no parameters but uses built-ins to interact with the user, similar to the no-parameters examples in Lecture 4.

1.2 How to Import Packages and Use Existing Functions

As discussed in Lecture 4, import packages using 'import library_name' or 'from library import function'. Examples include math, pandas, numpy, sklearn, and keras. After importing, call functions with library_name.function(arguments).

Example: Using math library (from Lecture 4).

```
import math

def use_math_functions():
    print(math.sqrt(25))  # Returns 5.0
    print(math.pow(2, 3))  # Returns 8.0

use_math_functions()
```

Other examples:
- import pandas as pd; df = pd.DataFrame([[1, 2], [3, 4]])
- import numpy as np; arr = np.array([1, 2, 3])
- from sklearn.linear$_{m}odelimportLinearRegression; model = LinearRegression()$
- from keras.models import Sequential; nn$_{m}odel = Sequential()$

These build on Lecture 4 by reusing the import syntax in functions.

1.3 How to Know the Existing Functions of a Library

To discover functions in a library, use built-in tools like dir(library_name) to list attributes and methods, or help(library_name) for documentation. This is useful after importing.

Example: Discovering math functions.

```
import math

def discover_math():
    print(dir(math))  # Lists all functions like ['acos', 'asin', 'sqrt', ...]
    help(math.sqrt)   # Shows documentation for sqrt

discover_math()
```

Use this to explore libraries mentioned in Lecture 4, like pandas or numpy.

2 Modules in Python

Modules are files containing Python code (functions, variables, classes) that can be imported and reused, promoting modularity as introduced in Lecture 4.

2.1 What are Modules

A module is a .py file with definitions. Built-in modules like math are pre-installed; custom modules are user-created. They extend functions by organizing code across files.

2.2 How to Create a Module

Save functions in a file like my_module.py. Then import it in another script.

Example: Create my_module.py with functions from Lecture 4.

Content of my_module.py:

```
def add(a, b, c):
    return a + b + c

def is_even_odd(num):
    if num % 2 == 0:
        return "Even"
    else:
        return "Odd"
```

To use it:

```
import my_module

sum_result = my_module.add(1, 2, 3)  # Returns 6
print(sum_result)

even_check = my_module.is_even_odd(4)  # Returns "Even"
print(even_check)
```

This connects to Lecture 4 by reusing user-defined functions in a module.

3 Data Structures in Python

Data structures store and organize data efficiently. Python's built-in ones include lists, tuples, dictionaries, and sets. Arrays are available via numpy. We'll explain each with simple programs, building on Lecture 4 functions.

3.1 Comparison of Data Structures

Before diving into each data structure, here is a comparison table summarizing their key characteristics and usage.

3.2 Lists

Lists are one of the most versatile data structures in Python. They are ordered collections, meaning elements have a specific position and can be accessed by index (starting from 0). Lists are mutable, allowing addition, removal, or modification of elements. They can contain duplicates and elements of different types (integers, strings, floats, etc.). Lists support slicing (e.g., my_list[1:3]), iteration with loops, and comprehension for creating new lists.

Common operations include: - Appending elements with `append()` - Inserting at a position with `insert()` - Removing with `remove()` or `pop()` - Sorting with `sort()` or `sorted()` - Reversing with `reverse()`

Example: Simple list operations.

```python
def list_demo():
    my_list = [1, 2, 3, "apple", 4.5]
    print(my_list[0])  # Access first element: 1
    my_list.append(6)  # Add element
    print(len(my_list))  # Length: 6
    print(my_list[1:4])  # Slicing: [2, 3, 'apple']

list_demo()
```

Connect to Lecture 4: Modify sum_even_numbers() to take a list parameter and use it.

Modified sum_even_numbers in my_module.py:

```python
def sum_even_numbers(numbers):
    total = 0
    for num in numbers:
        num = float(num)
        if num % 2 == 0:
            total += num
    return total
```

Usage:

```python
from my_module import sum_even_numbers

numbers = [1, 2, 3, 4, 5, 6]
even_sum = sum_even_numbers(numbers)
print(even_sum)  # 12.0
```

3.2.1 Additional List Examples

Example 1: Iterating over a list with mixed types.

```python
fruits = ["apple", "banana", "cherry", "dfsdfsfsd", "mrmrmr",
    2323, 232355.3434]
for fruit in fruits:
    print(fruit)
    print("Hi in For")

print("Hi out For")
```

Example 2: Creating a list using range.

```
y = list(range(10, 0, -2))  # countdown by 2
print(y)  # [10, 8, 6, 4, 2]
```

Example 3: Append, insert, remove operations.

```
numbers = [1, 2, 3, 4, 5]
print("Initial list:", numbers)  # Initial list: [1, 2, 3, 4,
    5]

numbers.append(10)
print(numbers)  # [1, 2, 3, 4, 5, 10]

numbers.insert(0, 100)
print(numbers)  # [100, 1, 2, 3, 4, 5, 10]

numbers.insert(3, 200)
print(numbers)  # [100, 1, 2, 200, 3, 4, 5, 10]

numbers.remove(200)
numbers.remove(100)
print(numbers)  # [1, 2, 3, 4, 5, 10]
```

Example 4: Pop operations.

```
list2 = [1, 2, 3, 5, 6, 9, 4, 70]
print(list2)  # [1, 2, 3, 5, 6, 9, 4, 70]

list2.pop()
list2.pop()
list2.pop()
print(list2)  # [1, 2, 3, 5, 6, 9]
```

Example 5: Sorting lists.

```
list3 = [100, 90, 10, 45, 400, 700, 12]
list4 = [100, 90, 10, 45, 400, 700, 12]

print(list3)  # [100, 90, 10, 45, 400, 700, 12]

list3.sort()  # Asc Order
print(list3)  # [10, 12, 45, 90, 100, 400, 700]

list4.sort(reverse=True)  # Desc order
print(list4)  # [700, 400, 100, 90, 45, 12, 10]
```

Example 6: Index and modifying elements.

```
my_list = [1, 2, 3, 4]
print(my_list)  # [1, 2, 3, 4]
```

```
try:
    index = my_list.index(5)  # 5 is not in the list
    print(index)
except ValueError:
    print("Value not found in the list")  # Value not found in the list

my_list[0] = 10
print(my_list)  # [10, 2, 3, 4]

my_list[2] = 30
print(my_list)  # [10, 2, 30, 4]
```

Example 7: Comprehensive list methods.

```
numbers = [1, 2, 3, 4, 5]

# Accessing elements
print("First item (numbers[0]):", numbers[0])  # 1
print("Second item (numbers[1]):", numbers[1])  # 2
print("Last item (numbers[-1]):", numbers[-1])  # 5
print("Second last item (numbers[-2]):", numbers[-2])  # 4
print("-------------------------------------------------")

# Append - adds 6 to the end
numbers.append(6)
print("After append(6):", numbers)  # [1, 2, 3, 4, 5, 6]
print("-------------------------------------------------")

# Insert - adds 6 at index 0
numbers.insert(0, 6)
print("After insert(0, 6):", numbers)  # [6, 1, 2, 3, 4, 5, 6]
print("-------------------------------------------------")

# Remove - removes the first occurrence of 6
numbers.remove(6)
print("After remove(6):", numbers)  # [1, 2, 3, 4, 5, 6]
print("-------------------------------------------------")

# Pop - removes the last item
popped = numbers.pop()
print("After pop():", numbers)  # [1, 2, 3, 4, 5]
print("Popped value:", popped)  # 6
print("-------------------------------------------------")

# Copy - creates a copy of the list
copied_list = numbers.copy()
print("Copied list:", copied_list)  # [1, 2, 3, 4, 5]
print("-------------------------------------------------")
```

```python

# Sort - sorts the list in ascending order
numbers.sort()
print("After sort():", numbers)  # [1, 2, 3, 4, 5]
print("------------------------------------------------")

# Reverse - reverses the list
numbers.reverse()
print("After reverse():", numbers)  # [5, 4, 3, 2, 1]
print("------------------------------------------------")

# Index - finds the position of the first occurrence of 3
if 3 in numbers:
    print("Index of 3:", numbers.index(3))  # 2
else:
    print("3 is not in the list.")
print("------------------------------------------------")

# Clear - removes all items
numbers.clear()
print("After clear():", numbers)  # []
print("------------------------------------------------")
```

3.3 Tuples

Tuples are similar to lists but immutable, meaning once created, their elements cannot be changed, added, or removed. This makes them suitable for fixed collections of items, like coordinates or constants. Tuples are faster and use less memory than lists. They support indexing, slicing, and unpacking (e.g., a, b = my_tuple).

Common uses: As keys in dictionaries (since immutable), returning multiple values from functions.

Example:

```python
def tuple_demo():
    my_tuple = (1, 2, "banana", 3.5)
    print(my_tuple[1])  # 2
    # my_tuple[0] = 0  # Error: immutable
    a, b, _, _ = my_tuple  # Unpacking
    print(a + b)  # 3

tuple_demo()
```

Connect: Use add() from Lecture 4 on tuple elements.

```python
from my_module import add

```

```
coords = (1, 2, 3)
total = add(*coords)  # Unpack tuple: 6
print(total)
```

3.3.1 Additional Tuple Examples

Example 1: Accessing and unpacking tuples.

```
x = (10, 14, 19)
print(x)  # (10, 14, 19)
print(x[0])  # 10
print(x[1])  # 14
print(x[2])  # 19

item1, item2, item3 = x
print(item1)  # 10
print(item2)  # 14
print(item3)  # 19

item1 = 100
print(item1)  # 100
```

Example 2: Converting tuple to list.

```
# Convert Tuple to list
my_tuple = (1, 2, 3, 4)
my_list = list(my_tuple)
print(my_list)  # [1, 2, 3, 4]

my_list[0] = 1000
print(my_list)  # [1000, 2, 3, 4]
```

Example 3: Comprehensive tuple operations.

```
# Creating a tuple
coordinates = (1, 2, 3)
print("Tuple:", coordinates)  # Tuple: (1, 2, 3)

# Accessing elements
print("First element:", coordinates[0])  # 1
print("Second element:", coordinates[1])  # 2
print("Third element:", coordinates[2])  # 3
print("-----------------------------------------")

# Trying to change a value (this will raise an error)
# coordinates[0] = 10  #  TypeError: 'tuple' object does not
   support item assignment

# Length of the tuple
```

```python
print("Length of tuple:", len(coordinates))  # 3
print("------------------------------------------")

# Unpacking the tuple into separate variables
x, y, z = coordinates
print("Unpacked values:")
print("x =", x)  # 1
print("y =", y)  # 2
print("z =", z)  # 3
print("----------------------------------------------------")

# Tuple with one item (note the comma)
single_item_tuple = (5,)
print("Single-item tuple:", single_item_tuple)  # (5,)
print("-----------------------------------------------------")

# Using tuple in a function return: the function return tuple (list of values to be returned)
def get_student_info():
    return ("Alice", 20, "Computer Science")

name, age, major = get_student_info()
print("\nStudent Info:")
print("Name:", name)  # Alice
print("Age:", age)  # 20
print("Major:", major)  # Computer Science
print("------------------------------------------")
```

3.4 Dictionaries

Dictionaries store data in key-value pairs, where keys are unique and immutable (e.g., strings, numbers, tuples), and values can be any type, including other data structures. Since Python 3.7, dictionaries preserve insertion order. They provide fast lookups, insertions, and deletions based on keys. Not indexed by numbers, but by keys.

Common methods: - Access with `dict[key]` or `get(key, default)` - Add/update with `dict[key] = value` or `update()` - Iterate with `keys()`, `values()`, `items()`

Example: Use dictionaries to store Kaggle dataset metadata and print in the CLI.

```python
def kaggle_metadata():
    dataset = {
        "name": "Titanic",
        "size": "1MB",
```

```python
        "rows": 891,
        "columns": 12,
        "description": "Passenger data for survival
    prediction"
    }
    print("Dataset Name: " + dataset["name"])
    print("Size: " + dataset.get("size", "Unknown"))
    print("Rows: " + str(dataset["rows"]))
    for key, value in dataset.items():
        print(key + ": " + str(value))

kaggle_metadata()
```

Connect: Compute grade average from dict of scores using compute_grade(). Assuming compute_grade takes a score parameter (from Lecture 4).

```python
from my_module import compute_grade

scores = {"Alice": 85, "Bob": 92}
for name, score in scores.items():
    print(name + ": " + compute_grade(score))
```

3.4.1 Additional Dictionary Examples

Example 1: Accessing dictionary values.

```python
# Create a dictionary to store customer information
customer = {
    "name": "John Smith",
    "age": 30,
    "is_verified": True
}

# Access existing keys
print("Customer name:", customer["name"])        # John Smith
print("Customer age:", customer["age"])          # 30
print("Is verified:", customer["is_verified"])   # True
print("------------------------------------------")
```

Example 2: Dictionary methods.

```python
customer = {
    "name": "John",
    "age": 25,
    "type": "silver"
}

# keys, values, and items
```

```python
print("Keys:", customer.keys())  # dict_keys(['name', 'age', 'type'])
print("Values:", customer.values())  # dict_values(['John', 25, 'silver'])
print("Items:", customer.items())  # dict_items([('name', 'John'), ('age', 25), ('type', 'silver')])
print("------------------------------------------")

# get and update
print("Membership type:", customer.get("type", "standard"))  # silver
customer.update({"type": "gold", "email": "john@example.com"})
print("Updated customer:", customer)  # {'name': 'John', 'age': 25, 'type': 'gold', 'email': 'john@example.com'}
print("------------------------------------------")

# pop and popitem
customer.pop("age")
print("After pop('age'):", customer)  # {'name': 'John', 'type': 'gold', 'email': 'john@example.com'}
print("------------------------------------------")

last_item = customer.popitem()
print("Last item removed:", last_item)  # ('email', 'john@example.com')
print("------------------------------------------")

# copy and clear
copy_customer = customer.copy()  # Copy dictionary
print("Copied dictionary:", copy_customer)  # {'name': 'John', 'type': 'gold'}
print("------------------------------------------")

customer.clear()  # CLEAR DICTIONARY
print("Cleared dictionary:", customer)  # {}
print("------------------------------------------")
```

3.5 Sets

Sets are collections of unique elements, with no duplicates allowed. They are unordered, so no indexing or slicing. Elements must be immutable. Sets are highly efficient for membership testing (in operator) and mathematical operations like union, intersection, difference, symmetric difference.

Common methods: - Add with `add()` - Remove with `remove()` or `discard()` - Set operations: `union()`, `intersection()`, etc.

Example:

```
def set_demo():
    my_set = {1, 2, 3, 2}  # Duplicates removed: {1,2,3}
    my_set.add(4)
    print(my_set)  # {1,2,3,4}
    print(2 in my_set)  # True
    other_set = {3, 4, 5}
    print(my_set.union(other_set))  # {1,2,3,4,5}

set_demo()
```

Connect: Check even/odd in a set using is_even_odd().

```
from my_module import is_even_odd

num_set = {1, 2, 3, 4}
for num in num_set:
    print(str(num) + " is " + is_even_odd(num))
```

3.6 Arrays and Other Data Structures

Arrays in Python are provided by the NumPy library, not built-in. They are designed for efficient storage and manipulation of numerical data, especially in scientific computing. Arrays are homogeneous (all elements same type), support multi-dimensional shapes (e.g., matrices), and enable vectorized operations without loops for speed.

Common operations: Arithmetic on entire arrays, slicing, reshaping with `reshape()`, aggregation like `sum()`, `mean()`.

Example:

```
import numpy as np

def array_demo():
    arr = np.array([1, 2, 3])
    print(arr * 2)  # [2 4 6]
    matrix = np.array([[1, 2], [3, 4]])
    print(matrix.shape)  # (2, 2)
    print(np.mean(arr))  # 2.0

array_demo()
```

Other structures: Strings (immutable sequences), Queues/Deques (from collections module for FIFO/LIFO).

4 Example Programs on Data Structures

Here are 10 programs, each using functions from Lecture 4 to connect concepts.

4.1 Program 1: List Sum Using add()

```
from my_module import add

def sum_list(lst):
    return add(lst[0], lst[1], lst[2])  # For 3 elements

print(sum_list([1, 2, 3]))  # 6
```

4.2 Program 2: Tuple Area Calculation

Assuming area from Lecture 4 rectangle example.

```
from my_module import area

dimensions = (5, 3)
print(area(*dimensions))  # 15.0
```

4.3 Program 3: Dictionary Even Check

```
from my_module import is_even_odd

ages = {"Alice": 20, "Bob": 21}
for name, age in ages.items():
    print(name + "'s age is " + is_even_odd(age))
```

4.4 Program 4: Set Grade Computation

Assuming compute$_g radetakesscore$.

```
from my_module import compute_grade

scores_set = {85, 92, 65}
for score in scores_set:
    print("Grade for " + str(score) + ": " + compute_grade(score))
```

4.5 Program 5: Numpy Array Sum Even

```python
import numpy as np
from my_module import sum_even_numbers

arr = np.array([1, 2, 3, 4])
print(sum_even_numbers(arr.tolist()))  # 6.0
```

4.6 Program 6: List Rectangle Areas

```python
from my_module import area

rects = [(5,3), (4,2), (6,1)]
for l, w in rects:
    print(area(l, w))
```

4.7 Program 7: Dictionary Kaggle Print with print$_{s}um()$

```python
from my_module import print_sum

metadata = {"rows": 891, "columns": 12, "features": 10}
print_sum(metadata["rows"], metadata["columns"], metadata["features"])
```

4.8 Program 8: Tuple Even Sum

```python
from my_module import sum_even_numbers

nums = (1, 2, 3, 4, 5, 6)
print(sum_even_numbers(list(nums)))  # 12.0
```

4.9 Program 9: Set Addition Using get$_{s}um()$

```python
from my_module import get_sum

my_set = {get_sum(), 7, 8}  # get_sum() returns 6
print(my_set)  # {6,7,8}
```

4.10 Program 10: Array Grade Average

```python
import numpy as np
from my_module import compute_grade

scores_arr = np.array([85, 92, 65])
avg = np.mean(scores_arr)
print("Average grade: " + compute_grade(avg))
```

5 Assignments

5.1 Assignment 1

Create a module with a function that takes a list of numbers and returns a dictionary with even/odd counts. Use is$_{e}ven_{o}dd()fromLecture4$.

5.2 Assignment 2

Write a program using a tuple to store rectangle dimensions, compute areas with area() from Lecture 4, and store results in a set.

5.3 Assignment 3

Use a dictionary for Kaggle dataset metadata (add at least 3 datasets), import numpy, and compute average rows using array operations.

5.4 Assignment 4

Build a function that takes a set of scores, uses compute$_{g}rade()toassigngrades,and$

Data Structure	Characteristics	How to Use It
List	Ordered, mutable (can change elements), allows duplicates, can hold different data types (heterogeneous).	Create: `my_list = [1, 'a', 2.5]` Access: `print(my_list[0])` Modify: `my_list.append(3)` Other methods: extend, insert, remove, pop, sort.
Tuple	Ordered, immutable (cannot change elements after creation), allows duplicates, heterogeneous. Faster and more memory-efficient than lists for fixed data.	Create: `my_tuple = (1, 'a', 2.5)` Access: `print(my_tuple[1])` No modification methods; used for unpacking or as keys in dicts.
Dictionary	Unordered (insertion order preserved since Python 3.7), mutable, stores key-value pairs, keys must be unique and immutable, values can be any type. Fast lookups.	Create: `my_dict = {'key': 'value'}` Access: `print(my_dict['key'])` Modify: `my_dict['new'] = 10` Methods: get, keys, values, items, update.
Set	Unordered, mutable, stores unique elements only (no duplicates), elements must be immutable. Fast membership testing and set operations.	Create: `my_set = {1, 2, 3}` Add: `my_set.add(4)` Operations: union (`|`), intersection (`&`), difference (`-`).
Array (NumPy)	Ordered, mutable, homogeneous (all elements same type, usually numbers), supports multi-dimensional, efficient for numerical computations and vectorized operations.	Import: `import numpy as np` Create: `arr = np.array([1, 2, 3])` Operations: `print(arr + 1)` (broadcasting), slicing, reshaping, mathematical functions.

Table 1: Comparison of Python Data Structures