Discussion-8 Python List and Dictionary (Dict) Data Types

Discussion Topic:

The list and dictionary object types are two of the most important and often used types in a Python program.

- What are some ways to insert, update, and remove elements from lists and dictionaries?
- Why would you choose one data type over another?
- Provide code examples demonstrating the usages of both data types.

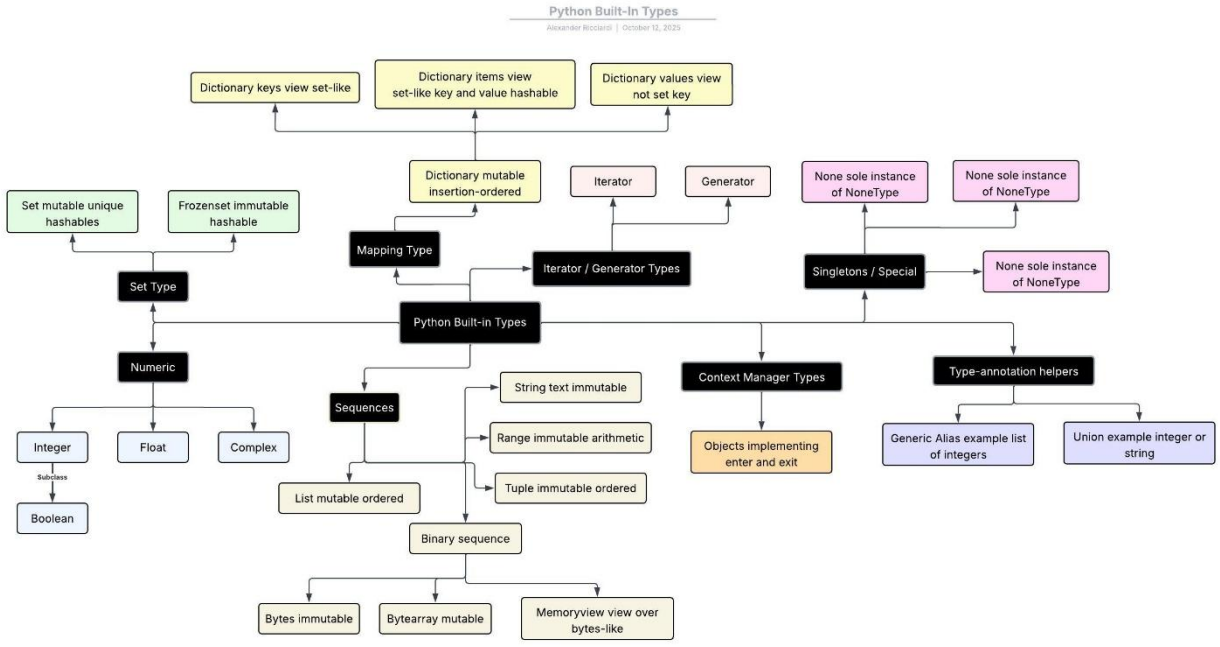
Actively participate in this discussion by providing constructive feedback on the criteria, rationales, and examples posted by your peers. Include additional code examples in your responses if applicable.

My Post:

Hello Class,

In Python, the `List` and Dictionary (`Dict`) are the most commonly used data types to organize and hold data within a program (Buruiană, 2024).
Both `List` and `Dict` are built-in data types, see Figure 1:

Figure 1
Python Built-In Data Type



Note: The diagram illustrates the hierarchical taxonomy of Python's built-in data types. Data from "Built-in Type – Python 3.14.0 Documentation" (Python, n.d.)

List

As shown in Figure 1, the List data type is a subclass of the Sequences data type. It is an ordered index-based mutable sequence typically used to store collections of homogeneous items; however, a `List` can store heterogeneous items such as a mix of `numeric` and `string` data types. The List class supports various operations, see Table 1:

Table 1

List Operations

Operation	Description	Example
`append(x)`	Adds an item to the end of the list	`list.append('orange')`
`extend(iterable)`	Extends the list by appending all items from the iterable data type	`list.extend(['grape', 'kiwi'])`
`insert(i, x)`	Inserts an item at a given index (index i)	`list.insert(1, 'mango')`
`remove(x)`	Removes the first item whose value equals x	`list.remove('banana')`
`pop([i])`	Removes and returns the item at the given position; removes the last item if no index is specified	`list.pop()` or `list.pop(1)`
`clear()`	Removes all items from the list	`list.clear()`
`index(x[, start[, end]])`	Returns the index of the first item whose value equals x	`list.index('banana')`
`count(x)`	Returns the number of times x appears in the list	`list.count('apple')`
`sort(key=None, reverse=False)`	Sorts the items in place	`list.sort()`
`reverse()`	Reverses the elements in place	`list.reverse()`
`copy()`	Returns a shallow copy of the list	`new_list = list.copy()`

Note: The table provides a list, descriptions, and syntax examples of the various operations supported by the `List` class data type. Data from "Built-in Type – Python 3.14.0 Documentation" (Python, n.d.)

A `List` is great when item order is important, duplicates are allowed, and positional operations (e.g., iterate, sort, slice) are required or needed. They are also great for data manipulation.

For instance, a `List` is the ideal data structure for the implementation of a to-do list. The example below illustrates a simple to-do implementation.

```
# create todo list
```

```
todo_list = ["Buy groceries", "Walk the dog", "Pay bills"]
print("Initial Todo List:", todo_list)
print()

# append(), adds task to end
todo_list.append("Call dentist")
print("After append():", todo_list)

# insert(), adds task at specific index
todo_list.insert(0, "Morning meditation")
print("After insert(0):", todo_list)
print()


# Modify, direct indexing
todo_list[1] = "Buy groceries and fresh vegetables"
print("After updating index 1:", todo_list)
print()

# remove(), using a value
todo_list.remove("Pay bills")
print("After remove():", todo_list)

# pop(), remove - using index - and return last task
completed = todo_list.pop()
print(f"Completed: '{completed}'")
print("After pop():", todo_list)
print()

# Check if task exists
if "Walk the dog" in todo_list:
    print("'Walk the dog' is in the list")

# Count total tasks
print(f"Total tasks remaining: {len(todo_list)}")
print()
```

Output:

```
Initial Todo List: ['Buy groceries', 'Walk the dog', 'Pay bills']

After append(): ['Buy groceries', 'Walk the dog', 'Pay bills', 'Call dentist']
After insert(0): ['Morning meditation', 'Buy groceries', 'Walk the dog', 'Pay bills', 'Call
dentist']
```

```
After updating index 1: ['Morning meditation', 'Buy groceries and fresh vegetables', 'Walk the dog', 'Pay bills', 'Call dentist']

After remove(): ['Morning meditation', 'Buy groceries and fresh vegetables', 'Walk the dog', 'Call dentist']
Completed: 'Call dentist'
After pop(): ['Morning meditation', 'Buy groceries and fresh vegetables', 'Walk the dog']

'Walk the dog' is in the list
Total tasks remaining: 3
```

Dict (Dictionary)

As shown in Figure 1, the `Dict` data type is a subclass of the `Mapping` data type. A hash map `key-value` pairs. It is a mutable, insertion-ordered mapping from unique, hashable keys to arbitrary value data types. Note that Dictionaries were unordered in earlier versions of Python, and since Python 3.7+, they are insertion-ordered. The keys must be hashable (immutable types like `str`, `int`, `tuple`, etc.); on the other hand, the value can be heterogeneous, that is, they can be a mix of strings, numbers, lists, custom objects, or even other `Dict`s. The `Dict` class supports various operations, see Table 2:

Operation	Description	Example
`clear()`	Removes all items from the dictionary	`dict.clear()`
`copy()`	Returns a shallow copy of the dictionary	`new_dict = dict.copy()`
`get(key[, default])`	Returns the value for key if key is in dictionary, else returns default (or None)	`dict.get('age', 0)`
`items()`	Returns a view object containing dictionary's key-value pairs as tuples	`dict.items()`
`keys()`	Returns a view object containing dictionary's keys	`dict.keys()`
`values()`	Returns a view object containing dictionary's values	`dict.values()`
`pop(key[, default])`	Removes key and returns its value; returns default if key not found	`dict.pop('age')`
`popitem()`	Removes and returns the last inserted key-value pair as a tuple	`dict.popitem()`
`setdefault(key[, default])`	Returns value of key if in dictionary; if not, inserts key with default value	`dict.setdefault('year', 'Junior')`

`update([other])`	Updates the dictionary with key-value pairs from other dictionary or iterable	`dict.update({'major': 'CS'})`
`fromkeys(seq[, value])`	Creates a new dictionary with keys from seq and values set to value	`dict.fromkeys(['a','b'], 0)`

Note: The table provides a list, descriptions, and syntax examples of the various operations supported by the `Dict` class data type. Data from "Built-in Type – Python 3.14.0 Documentation" (Python, n.d.)

A `Dict` is great for fast value look, `O(1)`, for associated items such as id (`key`) to customer info. object (`value`). Note that Python treats objects returned by `dict.keys()`, `dict.values()`, and `dict.items()` (key-value pair) as view objects. A `Dict` is also great for many other tasks, but it shines when items need to be identified by a name/label rather than by position (index).

For instance, a `Dict` is the ideal data structure for managing student profiles. The example below illustrates a simple student profile management implementation.

```
# Create a dictionary
student = {"id": 101, "name": "Ada Lovelace", "year": "Sophomore"}

# get(key[, default]), it is safer to read than student['gpa'] in case a field is missing
# e.g., if 'gpa' is missing, fall back to 0.0 instead of KeyError
gpa = student.get("gpa", 0.0) # --> 0.0

# setdefault(key[, default])
# it creates a courses list only if none is present, then appends to a course
student.setdefault("courses", []).append("CSC300")
student.setdefault("courses", []).append("MATH350")
# --> student["courses"] == ["CS101", "MATH201"]

# update([other]) adds new data or overrides existing data
student.update({"year": "Junior", "major": "CS"})  # --> year updated, major added

# fromkeys(seq[, value]) creates items with A, B, ..., F keys paired with the value 0
#  (e.g., grade counts) quickly
grade_dict = dict.fromkeys(["A", "B", "C", "D", "F"], 0)  # -->
{'A':0,'B':0,'C':0,'D':0,'F':0}
# Adds into the student record the grades dictionary
student["grades"] = grade_dict

# keys() / values() / items(), view objects
all_keys = list(student.keys())    # --> Keys ==
['id','name','year','major','courses','grade_counts']
all_vals = list(student.values())  # --> values
pairs    = list(student.items())    # --> list of (key, value) item -> tuples
```

```
# copy() takes a shallow snapshot of the dictionary
before_mod = student.copy()

# pop(key[, default]) removes a value based on a Key, and returns the removed value
declared_major = student.pop("major", None) # --> returns 'CS'

# popitem() removes the most recently added item (LIFO), works like a stack data type
last_key, last_val = student.popitem()  # --> removes 'grade_counts' the last inserted item

# clear() wipes the dictionary
student.clear()  # --> student == {}
```

The `List` and `Dict` data type have their own benefits and advantages, but as a rule of thumb:

- The `List` data type should be used for ordered sequences, data that needs to be manipulated based on position (index).
- The `Dict` data type should be used when items need names/keys, and fast access based on those keys is required or needed.

-Alex

References:

Buruiană, C. (2024, September 25). *Difference between list and dict in Python*. Enki. https://www.enki.com/post/list-and-dict-in-python

Pytrhon (n.d.). *Built-in types - Python 3.14.0 documentation.* Python. https://docs.python.org/3/library/stdtypes.html#