TOP

Data Structures with Python

- Dictionaries, Maps, and Hash Tables
    - dict: Your Go-to Dictionary
    - collections.OrderedDict: Remember the Insertion Order of Keys
    - collections.defaultdict: Return Default Values for Missing Keys
    - collections.ChainMap: Search Multiple Dictionaries as a Single Mapping
    - types.MappingProxyType: A Wrapper for Making Read-Only Dictionaries
- Array Data Structures
    - list: Mutable Dynamic Arrays
    - tuple: Immutable Containers
    - array.array: Basic Typed Arrays
    - str: Immutable Arrays of Unicode Characters
    - bytes: Immutable Arrays of Single Bytes
    - bytearray: Mutable Arrays of Single Bytes
- Records, Structs, and Data Transfer Objects
    - Simple Data Objects (`dict`)
    - Immutable Groups of Objects (`tuple`)
    - Custom Class
    - Data Classes (`dataclasses.dataclass`)
    - Convenient Data Objects (`collections.namedtuple`)
    - Improved Namedtuples (`typing.NamedTuple`)
    - Serialized C Structs (`struct.Struct`)
    - Fancy Attribute Access (`types.SimpleNamespace`)
- Sets and Multisets
    - Set: Your Go-to Set

Python Console

    - Frozenset: Immutable Sets
    - Collections.Counter: Multisets
- Stacks-(LIFO)
    - List: Simple, Built-in Stacks
    - Collections.deque: Fast and Robust Stacks
    - Queue.LifoQueue: Locking Semantics for Parallel Computing
- Queue-(FIFO)
    - List: Terribly Sloooow Queues
    - Collections.deque: Fast and Robust Queues
    - Queue.Queue: Locking Semantics for Parallel Computing
    - Multiprocessing.Queue: Shared Job Queues
- Priority Queues
    - List: Manually Sorted Queues
    - Heapq: List-Based Binary Heaps
    - Queue.PriorityQueue: Beautiful Priority Queues
- Conclusion: Python Data Structures

Dictionaries, Maps, and Hash Tables

Dictionaries in Python are a fundamental data structure that stores an arbitrary number of objects, each identified by a `unique key`. They are also known by various other names, such as `maps`, `hashmaps`, `lookup tables`, or `associative arrays`.

Key Characteristics

- **Efficient Lookups**: Allows for the quick retrieval of information associated with a given key.
- **Insertion and Deletion**: Supports the efficient insertion and deletion of objects.
- **Unique Keys**: Each key in a dictionary is unique, ensuring that every stored object can be quickly and uniquely identified.

Python Console

```
my_dict = {
    'name': 'John',
    'age': 25,
    'city': 'New York'
}
```

1. `dict`: Your Go-to Dictionary

Description:
Standard dictionary in Python, which stores key-value pairs.

Usage:
Efficient for fast lookups, inserts, updates, and deletions.

Example:

```
my_dict = {'apple': 1, 'banana': 2}
print(my_dict['apple'])  # Output: 1
```

2. collections.OrderedDict: Remember the Insertion Order of Keys

Description: A dictionary subclass that maintains the order in which keys were first inserted.

Usage: Useful when the order of items matters.

Example:

```
from collections import OrderedDict

od = OrderedDict()
od['apple'] = 1
od['banana'] = 2
for key in od:
    print(key)  # Output: apple, banana
```

Python Console

3. collections.defaultdict: Return Default Values for Missing Keys

TOP

Description: A dictionary subclass that provides default values for missing keys.

Usage: Useful for counting, grouping, or accumulating values.

Example:

```
from collections import defaultdict

dd = defaultdict(int)
dd['apple'] += 1
print(dd['apple'])  # Output: 1
print(dd['banana'])  # Output: 0 (default int value)
```

4. collections.ChainMap: Search Multiple Dictionaries as a Single Mapping

Description: Combines multiple dictionaries into a single view.

Usage: Useful for managing nested scopes or combining configurations.

Example:

```
from collections import ChainMap

dict1 = {'apple': 1}
dict2 = {'banana': 2}
chain = ChainMap(dict1, dict2)
print(chain['apple'])  # Output: 1
print(chain['banana'])  # Output: 2
```

Python Console

5. types.MappingProxyType: A Wrapper for Making Read-Only Dictionaries

Description: Provides a read-only view of a dictionary.

Usage: Useful for creating immutable dictionaries.

Example:

```
from types import MappingProxyType

original_dict = {'apple': 1}
proxy_dict = MappingProxyType(original_dict)
print(proxy_dict['apple'])  # Output: 1
# proxy_dict['banana'] = 2  # Raises TypeError
```

Array Data Structures in Python

An array is a fundamental data structure available in most programming languages, and it has a wide range of uses across different algorithms.

In this section, you'll take a look at array implementations in Python that use only core language features or functionality that's included in the Python standard library. You'll see the strengths and weaknesses of each approach so you can decide which implementation is right for your use case.

Use Cases:

- Use list for general-purpose ordered collections.
- Use tuple when an immutable ordered collection is needed.
- Use array.array for memory-efficient storage of homogeneous data.
- Use str for textual data.
- Use bytes and bytearray for binary data.

Array Data Structures

TOP

list: Mutable Dynamic Arrays

- **Characteristics**:
  - Ordered collection of elements.
  - Allows duplicate elements.
  - Elements can be of different data types.
  - Supports dynamic resizing.
- **Usage**:

```
my_list = [1, 2, 3, "hello"]
```

- **Common Methods:** `append()`, `extend()`, `insert()`, `remove()`, `pop()`, `sort()`, `reverse()`

tuple: Immutable Containers

- **Characteristics:**
  - Ordered collection of elements.
  - Allows duplicate elements.
  - Elements can be of different data types.
  - Once created, elements cannot be changed (immutable).
- **Usage**:

```
my_tuple = (1, 2, 3, "hello")
```

- **Common Methods:** `count()`, `index()`

Python Console

array.array: Basic Typed Arrays

TOP

- **Characteristics:**
    - Ordered collection of elements of the same type.
    - More memory-efficient than lists for large arrays of uniform data types.
    - Requires importing the array module.
- **Usage:**

```
from array import array
        my_array = array('i', [1, 2, 3, 4])
```

- **Common Methods:** `append()`,`extend()`, `insert()`, `remove()`,`pop()`

str: Immutable Arrays of Unicode Characters

- **Characteristics:**
    - Ordered collection of Unicode characters.
    - Immutable, meaning elements cannot be changed after creation.
- **Usage**:

```
my_str = "hello"
```

- **Common Methods:** `upper()`, `lower()`,`find()`,`replace()`, `split()`,`join()`

bytes: Immutable Arrays of Single Bytes

- **Characteristics:**

Python Console

- Ordered collection of bytes.
- Immutable, meaning elements cannot be changed after creation.
- Used for binary data.

- **Usage:**

```
my_bytes = b'hello'
```

- **Common Methods:** Similar to str, but adapted for bytes.

bytearray: Mutable Arrays of Single Bytes

Characteristics:

- Ordered collection of bytes.
- Mutable, meaning elements can be changed after creation.
- Used for binary data.

- **Usage:**

```
my_bytearray = bytearray(b'hello')
```

- **Common Methods:** Similar to bytes, with additional methods for mutability like append(), extend(), insert(), remove(), pop()

Records, Structs, and Data Transfer Objects

Python offers several data types that you can use to implement records, structs, and data transfer objects. In this section, you'll get a quick look at each i Python Console its unique characteristics. At the end, you'll find a summary and a decision-making guide that will help you make your own picks.

TOP

Simple Data Objects (`dict`)

Dictionaries are mutable and versatile data structures used to store key-value pairs.

```
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}

print(person["name"])  # Output: Alice
```

Immutable Groups of Objects (`tuple`)

Tuples are immutable sequences used to store a collection of items. Once created, their contents cannot be changed.

```
coordinates = (10, 20)
print(coordinates[0])  # Output: 10
```

Write a Custom Class: More Work, More Control

Creating a custom class provides more control over the data structure and behavior, allowing encapsulation and additional methods.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        return f"My name is {self.name} and I am {self.age} years
```

Python Console

```
person = Person("Alice", 30)
print(person.introduce())  # Output: My name is Alice and I am 30 years old.
```

TOP

Data Classes (`dataclasses.dataclass`)

Data classes, introduced in Python 3.7, simplify the creation of classes used primarily for storing data. They automatically add special methods like `__init__`, `__repr__`, and `__eq__`.

```
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int

person = Person("Alice", 30)
print(person)  # Output: Person(name='Alice', age=30)
```

Convenient Data Objects (`collections.namedtuple`)

Named tuples are a subclass of tuples that allow for named fields, improving code readability.

```
from collections import namedtuple

Person = namedtuple('Person', ['name', 'age'])
person = Person(name="Alice", age=30)

print(person.name)  # Output: Alice
```

Improved Namedtuples

Python Console

(`typing.NamedTuple`)

TOP

`typing.NamedTuple` provides a more robust way to define named tuples, allowing for type hints and more control.

```
from typing import NamedTuple

class Person(NamedTuple):
    name: str
    age: int

person = Person(name="Alice", age=30)
print(person.age)  # Output: 30
```

Serialized C Structs (`struct.Struct`)

The `struct` module allows for the conversion between Python values and C structs represented as Python bytes objects.

```
import struct

data = struct.pack('i5s', 42, b'Alice')
print(data)  # Output: b'*\x00\x00\x00Alice'
```

Fancy Attribute Access (`types.SimpleNamespace`)

`SimpleNamespace` provides a flexible way to create objects that can have arbitrary attributes added to them.

```
from types import SimpleNamespace

person = SimpleNamespace(name="Alice", age=30)
print(person.name)  # Output: Alice
```

Python Console

Sets and Multisets

A set is an unordered collection of objects that doesn’t allow duplicate elements. Typically, sets are used to quickly test a value for membership in the set, to insert or delete new values from a set, and to compute the union or intersection of two sets.

Set: Your Go-to Set

Sets are unordered collections of unique elements. They are mutable and provide efficient membership testing, union, intersection, and difference operations.

Characteristics:

- Unordered collection of unique elements.
- Mutable: elements can be added or removed.
- Supports operations like union, intersection, and difference.

Example

```
# Creating a set
fruits = {"apple", "banana", "cherry"}
print(fruits)  # Output: {'banana', 'apple', 'cherry'}
```

Frozenset: Immutable Sets

Frozensets are immutable versions of sets. Once created, their elements cannot be modified. This makes frozensets useful as dictionary keys or in other contexts where immutability is required.

Characteristics:

- Immutable: elements cannot be changed after creation.
- Can be used as a dictionary key or stored in other sets.

Python Console

Example

```python
# Creating a frozenset
immutable_fruits = frozenset({"apple", "banana", "cherry"})
print(immutable_fruits)  # Output: frozenset({'banana', 'apple', 'cherry'})

# Attempting to add an element raises an error
# immutable_fruits.add("orange")  # AttributeError

# Using frozenset as a dictionary key
frozen_set_dict = {immutable_fruits: "fruit set"}
print(frozen_set_dict)  # Output: {frozenset({'banana', 'apple', 'cherry'}): 'fruit s
```

Collections.Counter: Multisets

`collections.Counter` is a specialized dictionary subclass used to count hashable objects. It functions as a multiset, allowing multiple occurrences of elements and providing methods for common multiset operations.

Characteristics:

- Subclass of dict designed for counting hashable objects.
- Allows multiple occurrences of elements.
- Provides methods for common multiset operations like most common, subtracting, and combining counts.

Example

```python
from collections import Counter

# Creating a Counter
fruit_counts = Counter(["apple", "banana", "apple", "orange", "banana", "banana"])
print(fruit_counts)  # Output: Counter({'banana': 3, 'apple': 2, 'orange': 1})

# Accessing counts
print(fruit_counts["banana"])  # Output: 3

# Most common elements
print(fruit_counts.most_common(2))  # Output: [('banana', 3), ('apple', 2)]
```

Python Console

```python
# Subtracting counts
fruit_counts.subtract(["banana", "apple"])
print(fruit_counts)  # Output: Counter({'banana': 2, 'apple': 1, 'orange': 1})
```

TOP

Stacks (LIFOs)

A stack is a collection of objects that supports fast Last-In/First-Out (LIFO) semantics for inserts and deletes. Unlike lists or arrays, stacks typically don't allow for random access to the objects they contain. The insert and delete operations are also often called push and pop. Here is an overview of stack implementations in Python, focusing on lists, `collections.deque`, and `queue.LifoQueue`, along with their characteristics and examples.

List: Simple, Built-in Stacks

Characteristics:

- Lists are dynamic arrays and can be used as stacks using `append()` and `pop()`.
- Simple to implement and straightforward to use.

Example

```python
# Creating a stack using a list
stack = []

# Pushing elements onto the stack
stack.append(1)
stack.append(2)
stack.append(3)

print(stack)  # Output: [1, 2, 3]

# Popping an element from the stack
top = stack.pop()
print(top)    # Output: 3
print(stack)  # Output: [1, 2]
```

Python Console

Collections.deque: Fast and Robust Stacks

TOP

Characteristics:

- `deque` (double-ended queue) is part of the `collections` module.
- Provides O(1) time complexity for append and pop operations.
- Suitable for scenarios requiring fast stack operations and can also be used as a queue.

```
from collections import deque

s = deque()
s.append("eat")
s.append("sleep")
s.append("code")
print(s) # deque(['eat', 'sleep', 'code'])

s.pop() #'code'
s.pop() #'sleep'
s.pop() #'eat'
s.pop()

# Output:

# Traceback (most recent call last):
#  File "", line 1, in
#  IndexError: pop from an empty deque
```

Queue.LifoQueue: Locking Semantics for Parallel Computing

Characteristics:

- `LifoQueue` is part of the `queue` module and is thread-safe.

Python Console

- Provides methods that ensure safe concurrent access to the stack.
- Ideal for multi-threaded applications where thread safety is a concern.

TOP

```
from queue import LifoQueue
s = LifoQueue()
s.put("eat")
s.put("sleep")
s.put("code")

s
# <queue.LifoQueue object at 0x108298dd8>

s.get()
# 'code'
s.get()
# 'sleep'
s.get()
# 'eat'

s.get_nowait()
queue.Empty

s.get()  # Blocks/waits forever...
```

Queues (FIFOs)

A queue is a collection of objects that supports fast FIFO semantics for inserts and deletes. The insert and delete operations are sometimes called enqueue and dequeue. Unlike lists or arrays, queues typically don’t allow for random access to the objects they contain.Here is an overview of queue implementations in Python, focusing on lists, `collections.deque`, `queue.Queue`, and `multiprocessing.Queue`, along with their characteristics.

List: Terribly Sloooow Queues

Characteristics:

Python Console

- Lists can be used as queues using `append()` and `pop(0)`, but this results in O(n) time complexity for pop operations.
- Not recommended for large datasets due to performance issues.

```
q = []
q.append("eat")
q.append("sleep")
q.append("code")

q
# ['eat', 'sleep', 'code']

# Careful: This is slow!
q.pop(0)
# 'eat'
```

Collections.deque: Fast and Robust Queues

Characteristics:

- `deque` (double-ended queue) is part of the `collections` module.
- Provides O(1) time complexity for both append and pop operations, making it suitable for queue implementation.
- Supports both stack and queue functionalities.

```
from collections import deque
q = deque()
q.append("eat")
q.append("sleep")
q.append("code")

q
# deque(['eat', 'sleep', 'code'])

q.popleft()
# 'eat'
q.popleft()
```

Python Console

```
# 'sleep'
q.popleft()
# 'code'

q.popleft()

# Output:

Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
#  IndexError: pop from an empty deque
```

Queue.Queue: Locking Semantics for Parallel Computing

Characteristics:

- `Queue` is part of the `queue` module and provides thread-safe operations.
- Ensures that only one thread can access the queue at a time, making it ideal for multi-threaded applications.
- Offers methods for blocking and non-blocking operations.

```
from queue import Queue
q = Queue()
q.put("eat")
q.put("sleep")
q.put("code")

q
# <queue.Queue object at 0x1070f5b38>

q.get()
# 'eat'
q.get()
# 'sleep'
q.get()
# 'code'

q.get_nowait()
queue.Empty
```

Python Console

```
q.get()  # Blocks/waits forever...
```

TOP

Multiprocessing.Queue: Shared Job Queues

Characteristics:

- `multiprocessing.Queue` is designed for sharing data between processes.
- Supports multiple producers and consumers in a safe manner.
- Suitable for inter-process communication in parallel computing scenarios.

```
from multiprocessing import Queue
q = Queue()
q.put("eat")
q.put("sleep")
q.put("code")

q
# <multiprocessing.queues.Queue object at 0x1081c12b0>

q.get()
# 'eat'
q.get()
# 'sleep'
q.get()
# 'code'

q.get()  # Blocks/waits forever...
```

Priority Queues

A priority queue is a container data structure that manages a set of records with totally-ordered keys to provide quick access to the record with the smallest
the set.pPiority queue as a modified queue. Instead of retrieving the

Python Console

insertion time, it retrieves the highest-priority element. The priority of individual elements is decided by the order applied to their keys. Here is an overview of priority queue implementations in Python, focusing on lists, `heapq`, and `queue.PriorityQueue`, along with a summary of priority queues in Python.

List: Manually Sorted Queues

Characteristics:

- Lists can be used to implement priority queues by maintaining a sorted order.
- Insertion can be O(n) due to the need to keep the list sorted.
- Not efficient for large datasets or frequent priority updates.

```
q = []
q.append((2, "code"))
q.append((1, "eat"))
q.append((3, "sleep"))
# Remember to re-sort every time a new element is inserted,
# or use bisect.insort()
q.sort(reverse=True)

while q:
  next_item = q.pop()
  print(next_item)

# Output:

  (1, 'eat')
  (2, 'code')
  (3, 'sleep')
```

Heapq: List-Based Binary Heaps

Characteristics:

- `heapq` is a module that provides an efficient implementation of the min-heap algorithm.

Python Console

- Supports O(log n) time complexity for insertions and deletions.
- Can be used to create priority queues with minimal effort and maximum efficiency.

```
q = []
q.append((2, "code"))
q.append((1, "eat"))
q.append((3, "sleep"))
# Remember to re-sort every time a new element is inserted,
# or use bisect.insort()
q.sort(reverse=True)

while q:
  next_item = q.pop()
  print(next_item)

# Output:

(1, 'eat')
(2, 'code')
(3, 'sleep')
```

Queue.PriorityQueue: Beautiful Priority Queues

Characteristics:

- `PriorityQueue` is part of the `queue` module and provides a thread-safe implementation of priority queues.
- Offers blocking and non-blocking operations, suitable for multi-threaded applications.
- Internally uses a heap structure for efficient element retrieval based on priority.

```
from queue import PriorityQueue
q = PriorityQueue()
q.put((2, "code"))
q.put((1, "eat"))
q.put((3, "sleep"))
```

Python Console

```
while not q.empty():
    next_item = q.get()
    print(next_item)

Output:
  (1, 'eat')
  (2, 'code')
  (3, 'sleep')
```

Conclusion: Python Data Structures

In summary, Python offers a rich set of data structures, each designed to address specific needs and use cases. Understanding these structures is essential for writing efficient and effective code. Here's a brief recap of the main categories discussed:

- **Dictionaries, Maps, and Hash Tables**: These structures provide versatile key-value storage options, with features like order preservation and default values.
- **Array Data Structures**: From mutable lists to immutable tuples and specialized arrays, Python supports various ways to handle collections of elements efficiently.
- **Records, Structs, and Data Transfer Objects**: Different methods for organizing related data enable better management of complex information.
- **Sets and Multisets**: Useful for handling unique elements and counting occurrences, sets are foundational for many algorithms.
- **Stacks (LIFOs)**: Essential for last-in-first-out operations, stacks are implemented in multiple ways to suit different performance needs.
- **Queues (FIFOs)**: Crucial for first-in-first-out processing, queues support various concurrency and performance requirements.
- **Priority Queues**: These structures enable efficient retrieval of elements based on priority, with implementations suitable for various applica

Python Console

Interactive Python Console

TOP

Enter Python code...

Run

Python Console