coding temple

Python Intermediate

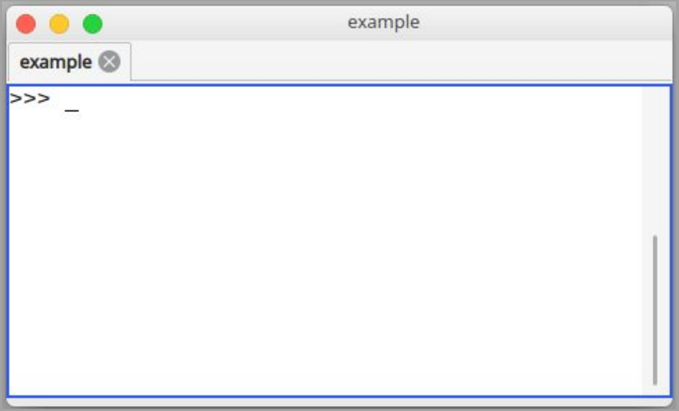
Module 3:

Python Tuples

coding temple

Intro

Hello, Coding Templers! 🌟
Join us on a thrilling journey into the enchanting realm of Python tuples, likened to exploring secrets of wisdom in an ancient library. This series serves as your guide to unravel the mysteries of these fascinating, immutable collections, from crafting basics to practical applications. We'll explore their immutable nature, performance benefits, and how they safeguard data integrity. Discover the artistry of manipulating tuples and learn when to choose them over lists. So, grab your favorite tea, find a cozy nook, and let's dive into this exciting chapter in our Python adventure. Whether you're a seasoned coder or a beginner, there's something for everyone in the captivating world of Python tuples.
Ready to explore, learn, and grow?
Happy coding! 📘✨

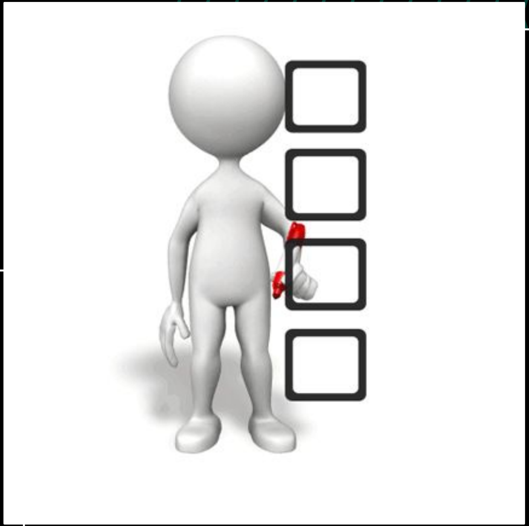


```
>>> _
```

Learning objectives

At the end of this lesson you should be able to:

- Create and identify the characteristics and properties of tuples in Python.
- Apply the concepts of positive and negative indexing to access individual elements within a tuple.
- Utilize slicing techniques to extract sub-sections of a tuple.
- Employ tuple unpacking methods and differentiate between the performance implications of using tuples versus lists.
- Demonstrate the use of count() and index() methods and perform concatenation of tuples using the + operator on tuples.
- Apply tuples effectively in function arguments and return values as well as recognize common errors and pitfalls when working with tuples.

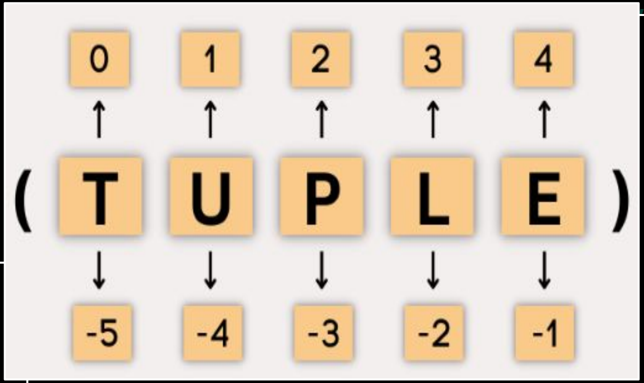


The Essence of Tuples

A Python tuple can be envisioned as a bookshelf in a majestic library, where each book (element) occupies a distinct position, and the order remains immutable. Similar to a series of books meticulously arranged in a specific sequence, creating a tuple ensures a fixed order, providing a clear understanding of the position of each element within the structure.

The Ordered World of Tuples

Tuples maintain a consistent order of elements. When you create a tuple, the sequence of its elements remains fixed and predictable. This order, similar to a well-organized world, enables you to rely on the arrangement of elements within the tuple. In other words, the first element stays first, the second remains second, and so forth. This inherent orderliness distinguishes tuples from other data structures and contributes to their reliability in preserving the sequence of information, making them particularly useful in scenarios where maintaining order is crucial.



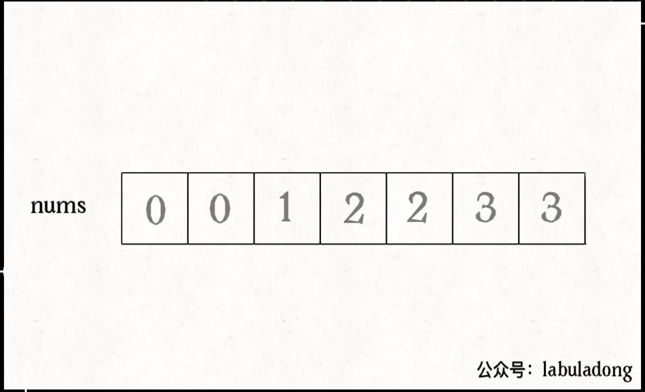
Immutable: The Unchanging Nature

Tuples in Python are immutable, much like the words in a classic novel that, once printed, remain unchanged. Once a tuple is created, its content cannot be modified, resembling a historical document preserved in a museum—unchanged, consistent, and enduring.

Just like a reference book that you trust to provide the same information every time you open it, a tuple gives you the certainty that its content will not change inadvertently.

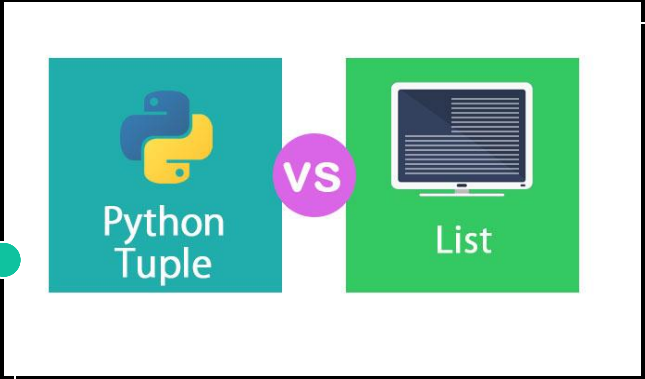
Duplicated and Tuples

Tuples in Python, similar to a book with repeated phrases or chapters, allow duplicates. This feature proves useful for scenarios where counting occurrences or ensuring specific values are consistently available is crucial, regardless of how many times they are repeated.



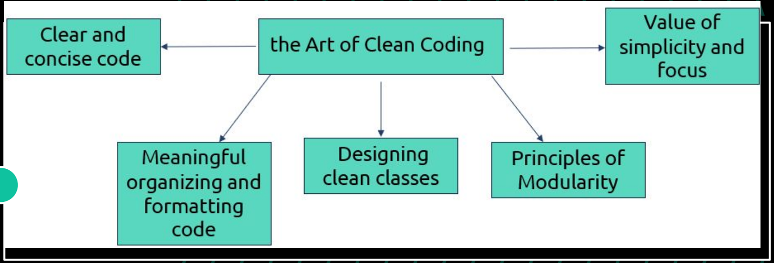
Why Choose Tuples Over Others?

When navigating the diverse world of Python collections, it's crucial to understand why and when to choose tuples over other data types like lists and dictionaries. Let's explore this decision in more depth.



Exploring the Art of Creating Tuples in Python

Think of creating a tuple as gathering a series of unique stories and binding them into one compelling anthology. Here's how it's done in Python.



Navigating Through the Chapters: Accessing Elements in Python Tuples

Hey there! Let's dive into the world of accessing elements in tuples, kind of like flipping through the pages and chapters of a book. 📚✨

Indexing: Finding Your Chapter.

How do indexing work?

Think of indexing as revisiting a familiar chapter in a different book within our grand Python library. Just like how you would find a specific chapter in a list-themed book by its chapter number (positive indexing) or start from the back and count backward (negative indexing), the same principles apply to tuples.

- **Positive index:** Each element in a tuple is indexed starting from 0.
- **Negative index:** Each element in a tuple is indexed starting from -1, which is the last element.

Let's watch a video to gain a more profound comprehension.

Indexing: Finding Your Chapter. (Cont.)

Here, accessing index 1 is like turning to the second chapter, titled "Adventure".

```
my_tuple = ("Prologue", "Adventure", "Epilogue")

print(my_tuple[1]) # Output: "Adventure"
```

Indexing: Finding Your Chapter. (Cont.)

This is like flipping to the last chapter, "Epilogue", in our book.

Let's click on the following video.

```
my_tuple = ("Prologue", "Adventure", "Epilogue")
print(my_tuple[-1]) # Output: "Epilogue"
```

Slicing Tuples: Skimming through Sections.

How does slicing work?

Slicing a tuple is similar to reading a specific section of a book, from one chapter to another. It's helpful to recall our previous adventures with lists. The process of slicing tuples is similar to skimming through specific sections of a book, much like we explored with lists. In both scenarios, we use a similar method to access a range of elements.

```
my_tuple = ("Prologue", "Adventure", "Epilogue")

print(my_tuple[0:2]) # Output: ("Prologue", "Adventure")
```

Advanced: Accessing Nested Tuples.

How does slicing work?

In nested tuples, akin to nested lists, accessing elements involves peeling back layers, and exploring stories within stories or books within books. This layered approach provides a familiar framework for understanding and navigating complex data structures. As you explore nested tuples, remember the interconnected tales from nested lists—a journey within the broader narrative of Python programming. 📘✨

Exploring a Story Within a Story Video

```
grand_library = ("Ancient Myths", ("Greek", "Norse"), "Modern Tales")
```

```
print(grand_library[1][1]) # Output: "Norse"
```

A Deeper Dive into Nested Collections Video

```
nested_anthology = ("Volume 1", ("Chapter 1", "Chapter 2", ("Page 1", "Page 2")), "Volume 2")
```

```
print(nested_anthology[1][2][1]) # Output: "Page 2"
```

Uncovering Layers in a Historical Record Video

```
historical_record = ("Medieval Era", ("Knights", "Castles", ("King", "Queen")), "Renaissance Period")
```

```
print(historical_record[1][2][1]) # Output: "Queen"
```

A Tale of Tuples and Lists Entwined Video

```
enchanted_library = ("Magic Tome", ["Ancient Scrolls", ("Spell", "Curse")], "Wizard's Guide")
```

```
print(enchanted_library[1][1][1]) # Output: "Curse"
```

Delving into Deeper Layers Video

```
mythical_collection = ("Greek Myths", [("Zeus", "Hera"), ["Mount Olympus",
("Lightning", "Thunder")]], "Norse Myths")
```

```
print(mythical_collection[1][1][1][1]) # Output: "Thunder"
```

A Tale with Maps and Encyclopedias Video

```
enchanted_library = ("Chapter 1", {"Mythical Creatures": ["Dragon", "Unicorn"],
"Legendary Places": ("Atlantis", "El Dorado")}, "Chapter 2")
```

```
print(enchanted_library[1]["Legendary Places"][1]) # Output: "El Dorado"
```

Workarounds for Modifying Tuples.

How does it work?

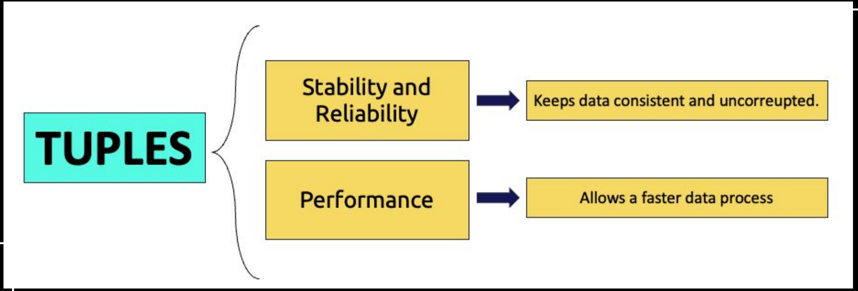
While you can't directly change a tuple, you can convert it into a list, modify the list, and then turn it back into a tuple. It's like photocopying a page from a book, making edits to the copy, and then binding it into a new book.

Let's watch a video to deepen our understanding.

```
my_tuple = ("Introduction", "Conclusion")
temp_list = list(my_tuple)
temp_list.append("Epilogue")
my_tuple = tuple(temp_list)
print(my_tuple) # Output: ("Introduction",
"Conclusion", "Epilogue")
```

Consequences and Benefits of Immutability

Exploring the dichotomy of immutability in programming reveals both the protective benefits of safeguarding data integrity and the consequential challenges of adaptability as follows:
Stability and Reliability: It ensures that the data remains consistent and uncorrupted over time.
Performance: Immutable objects, such as tuples, are faster to process than mutable counterparts like lists, akin to quickly referencing a well-known book versus editing a manuscript.


TUPLES
Stability and Reliability
Keeps data consistent and uncorreupted.
Performance
Allows a faster data process

coding temple

Unlocking the Secrets of Tuple Unpacking in Python

In this process, a tuple is compared to a treasure chest of stories, letting us efficiently extract and assign elements to variables.

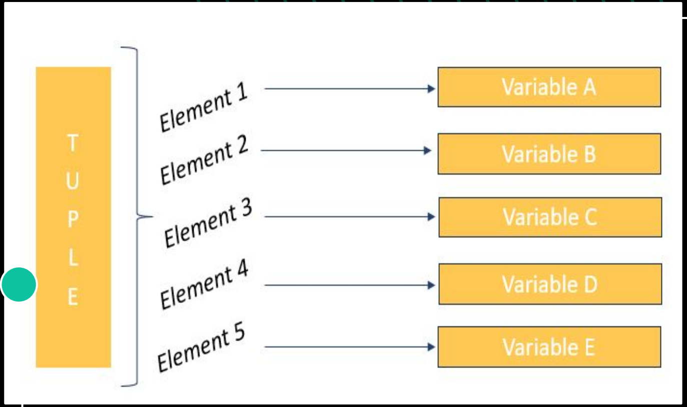
Let's explore the basics, Python 3's extended unpacking, and the versatile use of the asterisk (*) for added flexibility.

Basic Unpacking.

How does it work?

Basic unpacking involves assigning each element of a tuple to a separate variable. It's like taking a book with a collection of short stories and assigning each story to a different reader.

Let's watch a video to deepen our understanding.



Basic Unpacking Video

```
my_tuple = ("Magic", "Mystery", "Myth")
genre1, genre2, genre3 = my_tuple
print(genre1) # Output: "Magic"
print(genre2) # Output: "Mystery"
print(genre3) # Output: "Myth"
```

Extended Unpacking (Python 3.x Feature)

How does it work?

Python 3 introduced an extended form of unpacking that uses an asterisk (*) to capture multiple elements. It's like having a bag where you can put several stories together, leaving the rest individually.

This method is especially useful when dealing with tuples of unknown or variable length. It's akin to receiving a box of books of different sizes and being able to distribute them efficiently.

Let's watch a video to deepen our understanding.

Extended Unpacking (Python 3.x Feature) Video

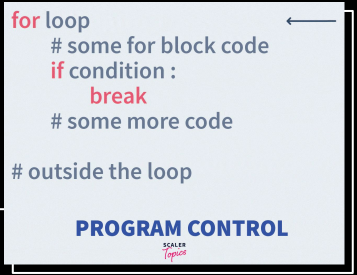
```
my_tuple = ("Prologue", "Adventure", "Climax", "Epilogue")
beginning, *middle, end = my_tuple
print(beginning) # Output: "Prologue"
print(middle) # Output: ["Adventure", "Climax"]
print(end) # Output: "Epilogue"
```

Navigating the Chapters:

Looping Through Tuples in Python

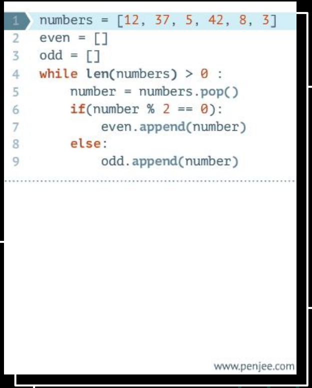
Looping through tuples in Python, is similar to how a reader journeys through the pages of a book. Tuples, with their unchanging nature, can be iterated over using for loops, while loops with indexing, and the convenient enumerate function. Join us as we unravel the art of traversing these Pythonic narratives!

FOR LOOPS



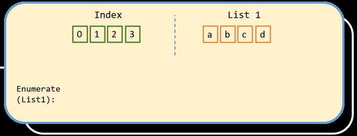
```
for loop
    # some for block code
    if condition :
        break
    # some more code

# outside the loop
```

WHILE LOOPS

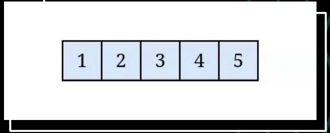


```
numbers = [12, 37, 5, 42, 8, 3]
even = []
odd = []
while len(numbers) > 0 :
    number = numbers.pop()
    if(number % 2 == 0):
        even.append(number)
    else:
        odd.append(number)
```

ENUMERATE



INDEXING



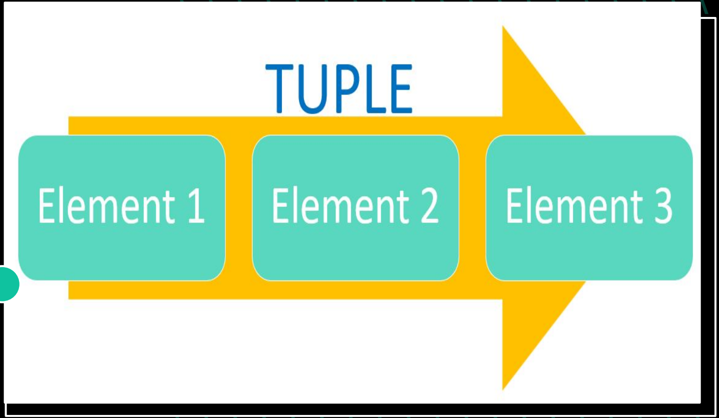
coding temple

Using the for Loop.

How does it work?

A `for` loop in Python is like reading through each chapter of a book sequentially. It's straightforward and efficient for going through each element in a tuple.

Let's watch a video to deepen our understanding.



Using the `for` Loop Video

```
book_titles = ("Moby Dick", "1984", "To Kill a Mockingbird")
for title in book_titles:
print(title)
```

Using the `while` Loop with Indexing.

How does it work?

A `while` loop, combined with indexing, is like flipping through the pages of a book one at a time. This method provides more control, as you can use an index to access each element.

Let's watch a video to deepen our understanding.

Using the `while` Loop with Indexing Video

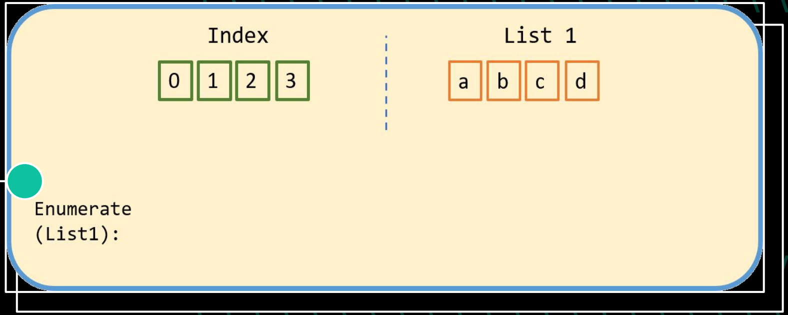
```python
authors = ("Herman Melville", "George Orwell", "Harper Lee")
index = 0
while index < len(authors):
      print(authors[index])
      index += 1
```

Looping with enumerate for Index and Element.

How does it work?

In Python's vast library, consider enumerate as a wise storyteller. When summoned, it not only narrates the tales (elements of a collection) but also meticulously notes their order (the index). What sets `enumerate` apart is its ability to return a tuple for each item in a collection, seamlessly combining the index and the item. While versatile and compatible with various Python collections, its true prowess shines in ordered structures like lists and tuples, where the index holds a clear and consistent significance.

Let's watch a video to deepen our understanding.



Looping with `enumerate` for Index and Element Video

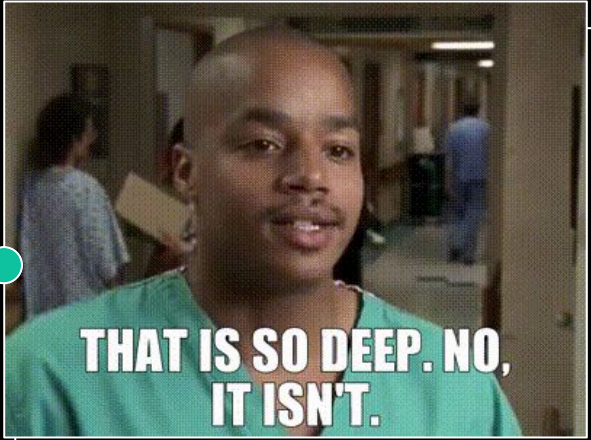
```
chapters = ("The Lighthouse", "The Ministry of Truth", "The Trial")
for index, chapter in enumerate(chapters):
print(f"Chapter {index + 1}: {chapter}")
```

coding temple

Navigating the Chapters:

Looping Through Tuples in Python

Navigating nested tuples in Python is akin to delving into a multi-layered novel, with each level representing a different layer of the story. Before embarking on the journey of looping through a nested tuple, understanding its depth—how many layers of stories it contains—is crucial. This awareness is essential for selecting the appropriate approach to navigate through the nested structure efficiently.



Why Depth Matters?

Understanding the depth of nested tuples is crucial; otherwise, you risk skimming the surface and missing deeper elements. In our lesson, we're exploring straightforward looping for tuples with a known depth. However, deeply nested tuples require advanced methods like recursion—akin to navigating an anthology with multiple layers of narrative.

Looping through Nested Tuples.

How does it work?

Looping through a nested tuple is like reading a book that contains other books. We explore each element layer by layer.

Looping Through Nested Tuples Video

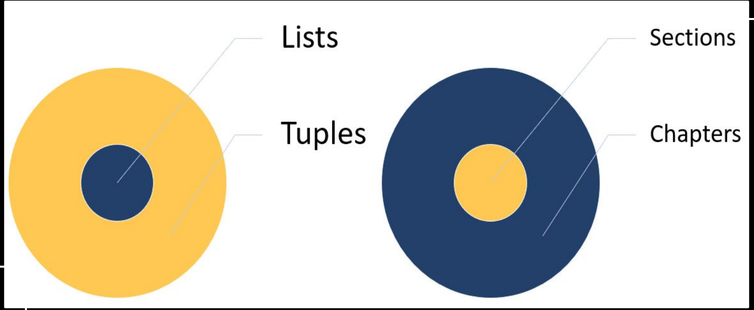
```
nested_tales = (("The Dawn", "The Noon"), ("The Dusk", "The Night"))
for pair in nested_tales:
    for tale in pair:
        print(tale)
```

Looping through Nested Tuples with Lists.

How does it work?

When tuples contain lists, it’s like a book with chapters that have their own sections. We loop through each chapter and then through its sections.

Let's watch a video to deepen our understanding.



Looping Through Tuples Nested with Lists Video

```
mixed_collection = ("Poetry", ["Sonnet", "Haiku"], "Prose")
for element in mixed_collection:
      if isinstance(element, list):
            for item in element:
                  print(f"List Item: {item}")
      else:
            print(f"Tuple Element: {element}")
```

Looping through Tuples with Dictionaries.

Tuples containing dictionaries are like volumes with detailed encyclopedic entries. We iterate through each entry for insights.

Let's tune in to a video to enhance our grasp of the concept.

Looping Through Tuples with Dictionaries Video

```
historical_records = ("Ancient", {"Rome": "Republic", "Greece": "Democracy"}, "Medieval")
for element in historical_records:
      if isinstance(element, dict):
            for key, value in element.items():
                  print(f"{key}: {value}")
      else:
            print(element)
```

Discovering the Tools of Tuples: Exploring Tuple Methods in Python

In our exploration of Python's tuple realm, similar to an immutable library, we now delve into essential tools: the count() and index() methods. Despite the limited methods of tuples, these functions act like a magnifying glass, aiding us in uncovering more from our tuple stories.

count(): Counting the Recurrences in a Tale.

How does it work?

The `count()` method in tuples is like counting how many times a specific word or theme appears in a book. It gives us the number of occurrences of a particular element.

Let's watch a video to improve our understanding of the concept.

```
literary_elements = ("Irony", "Metaphor",
"Irony", "Symbolism")
print(literary_elements.count("Irony")) #
Output: 2
```

index(): Locating the Chapter.

How does it work?

The `index()` method is similar to finding the page number of the first occurrence of a chapter title in a book. It returns the index of the first occurrence of a specified element.

Let's watch a video to improve our understanding of the concept.

```
book_chapters = ("Introduction", "Rising Action", "Climax",
"Conclusion")
print(book_chapters.index("Climax")) # Output: 2
```

The Symphony of Tuples:

Joining, Concatenating, and More in Python

In the rich tapestry of Python programming, akin to a library filled with diverse narratives, tuples play a crucial role. They are like individual stories or collections of tales, each with its distinct character. But what happens when we wish to merge these stories, to create an even grander narrative? Let's watch a video to improve our understanding of the concept.

Joining Tuples with the + Operator: Creating a Continuous Saga.

How does it work?

The + operator is the seamstress of our tuple fabric, seamlessly stitching together separate collections into one continuous thread.

Let's watch a video to improve our understanding of the concept.

```
epic = ("Odyssey", "Iliad")
drama = ("Hamlet", "Othello")
literary_union = epic + drama
print(literary_union)  # Output: ("Odyssey", "Iliad",
"Hamlet", "Othello")
```

coding temple

Preserving Structure with Nested Tuples: Maintaining the Integrity of Collections.

How does it work?

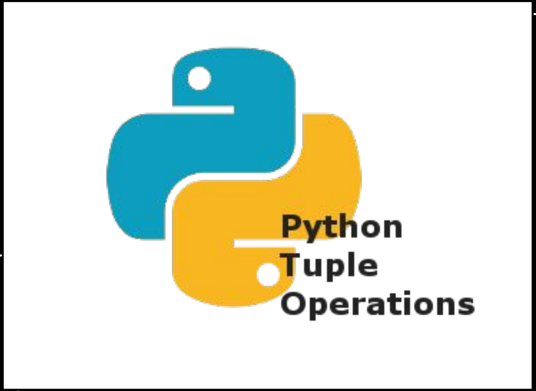
Nesting tuples allow us to combine collections while retaining their identity, like placing distinct sets of books on a shared shelf.

Let's watch a video to improve our understanding of the concept.

```
historical = ("War and Peace", "Gone with the
Wind")
fantasy = ("Lord of the Rings", "Harry Potter")
grand_library = (historical, fantasy)
print(grand_library) # Output: (("War and Peace",
"Gone with the Wind"), ("Lord of the Rings", "Harry
Potter"))
```

Exploring Other Tuple Operations: Beyond Joining

While the immutability of tuples limits certain operations, there are still various ways we can manipulate and explore these data structures such as Repeating Tuples and the Tuple Membership Test.



Exploring Other Tuple Operations: Beyond Joining - Repeating Tuples

How does it work?

Tuples can be repeated using the `times` operator, much like reprinting a beloved story to fill a shelf.

Let's watch a video to improve our understanding of the concept.

```
short_story = ("A Tale",)
anthology = short_story * 3
print(anthology) # Output: ("A Tale", "A
Tale", "A Tale")
```

Exploring Other Tuple Operations:

Beyond Joining - Tuple Membership

How does it work?

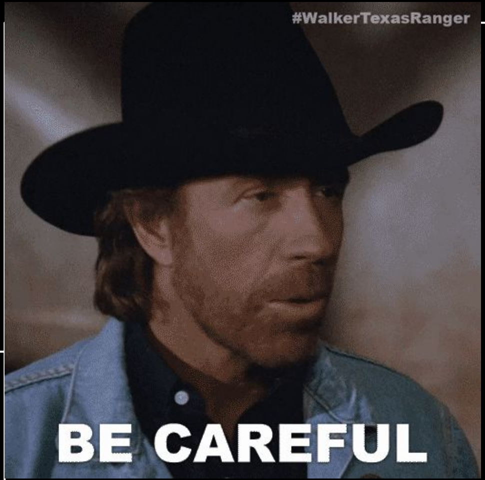
Checking if an item exists in a tuple is like searching for a specific book in a collection.

Let's watch a video to enhance our comprehension of the idea.

```
if "Odyssey" in literary_union:
print("Epic tale found!")
```

Navigating the Path: Best Practices and Pitfalls with Tuples in Python

As we continue our journey through the rich landscape of Python, it's crucial to navigate with wisdom and awareness. Tuples, like timeless tomes in our library of data structures, have their specific uses and cautionary tales.



In-Class Built-in functions Exercises: Hands-on Coding!

💡 **Hey there! Before we unveil our solutions**, how about taking a shot at it yourself? Keep in mind that we've provided some broad strokes, so a detailed approach will help fine-tune the solution. But, this is all in good fun and a chance for you to flex those problem-solving muscles. Give it a go, and remember, it's all about learning and improving!

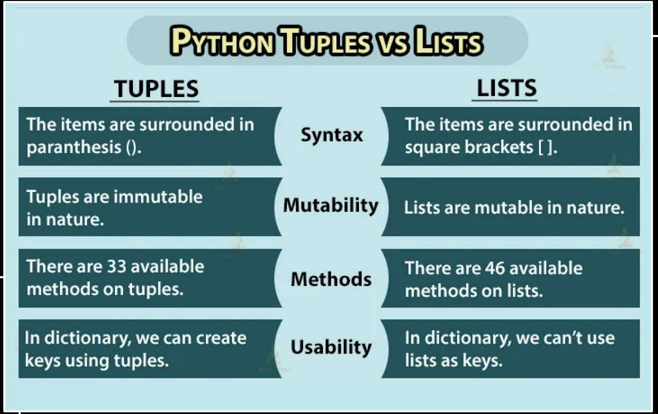
Let's **code** awesome things together!

The Closing Chapter: Embracing the Journey with Python Tuples

We've journeyed through the halls of immutable collections, uncovering the secrets and strengths of tuples, and now it's time to reflect on what we've learned:

- Create and access tuples from the basics to the more intricate practices of unpacking and looping through them.
- How tuples, like well-crafted tales, are an essential part of Python's storytelling.
- Join and concatenate tuples.
- How to weave separate stories into a greater narrative without altering their original essence.

In your Python journey, let tuples guide you as you explore the vast programming universe with confidence and curiosity. Keep writing scripts, debugging programs, and most importantly, savor every step in the enchanting Python world.



PYTHON TUPLES VS LISTS

TUPLES		LISTS
The items are surrounded in paranthesis ().	Syntax	The items are surrounded in square brackets [ ].
Tuples are immutable in nature.	Mutability	Lists are mutable in nature.
There are 33 available methods on tuples.	Methods	There are 46 available methods on lists.
In dictionary, we can create keys using tuples.	Usability	In dictionary, we can't use lists as keys.