**Lambda Functions:**

Lambda functions are anonymous functions in Python, used for creating small, one-time, and inline function objects.

**Syntax**: lambda arguments: expression

The expression is executed and the result is returned.

Simple Example:

Addition using lambda:

add = lambda x, y: x + y

print(add(5, 3)) # Output: 8

**Intermediate Uses** - Sorting and Filtering:

Sorting Example:

my\_list = [(1, 'banana'), (2, 'apple'), (3, 'orange')]

my\_list.sort(key=lambda x: x[1])

**Filtering** Example:

numbers = [1, 2, 3, 4, 5, 6]

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))

=========================================================================

**FILTER FUNCTION**

The `filter()` function in Python is used to construct an iterator from elements of an iterable for which a arbitrary function returns **True**.

Syntax:

filter(function, iterable)

- `function`: A function that tests if elements of an iterable return true or false. If None, the function defaults to the identity function, effectively filtering out false elements.

- `iterable`: An iterable like sets, lists, tuples, etc., whose elements are to be tested by the function.

The function needs to return a boolean value (True or False). `filter()` then applies this function to every element of the iterable and filters out the elements for which the function returns True.

You have already seen a Simple Example:

Advanced Use:

`filter()` can be used with more complex functions and iterables. For example, filtering objects based on their attributes:

class Product:

def \_\_init\_\_(self, name, price):

self.name = name

self.price = price

products = [Product("Apple", 1), Product("Banana", 2), Product("Cherry", 3)]

expensive\_products = filter(lambda p: p.price > 1, products)

expensive\_product\_names = [p.name for p in expensive\_products]

print(expensive\_product\_names) # Output: ['Banana', 'Cherry']

Notes:

- The result of `filter()` is an iterator, which is more memory-efficient than a list. To get a list, you can wrap it with `list()`.

- `filter()` is often used with `lambda` functions, but you can also define a regular function for more complex scenarios.

- For simple cases, `filter()` can be more readable and expressive compared to list comprehensions.

The `filter()` function is a convenient way to filter items in an iterable that meet a certain condition, enhancing code readability and expressiveness.

=========================================================================

**Advanced Uses - Map and Reduce**:

Map Example:

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

squared = list(map(lambda x: x\*\*2, numbers))

=========================================================================

MAP FUNCTION:  
  
The `map()` function in Python is a built-in function used to apply a specific function to each item of an iterable and return a map object, which is an iterator. It is commonly used for transforming data.

Syntax:

map(function, iterable, ...)

- `function`: A function that is applied to each item of the iterable. Each item is sent to the function as an argument.

- `iterable`: An iterable like a list, tuple, etc. Multiple iterables can be passed; the function should take as many arguments as there are iterables.

The result is a map object which can be converted into another iterable like a list or tuple.

Simple Example:

Suppose you have a list of numbers and you want to square each number:

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

squared = map(lambda x: x\*\*2, numbers)

print(list(squared)) # Output: [1, 4, 9, 16, 25]

**Advanced** **Use**:

`map()` can also be used with more than one iterable. The iterables must be of the same length.

Multiple Iterables Example:

numbers1 = [1, 2, 3]

numbers2 = [4, 5, 6]

summed = map(lambda x, y: x + y, numbers1, numbers2)

print(list(summed)) # Output: [5, 7, 9]

Notes:

- **The result of `map()` is a map object, an iterator.** To get a list or tuple, use `list()` or `tuple()`.

- `map()` is often used with `lambda` functions for simpler operations. For more complex transformations, a regular function can be defined.

- For simple transformations, `map()` can sometimes be more concise and readable compared to list comprehensions.

The `map()` function is especially useful in data transformation tasks, offering a concise way to apply a function to each item in an iterable.

=========================================================================

Reduce Example:

from functools import reduce

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

summed = reduce(lambda x, y: x + y, numbers)

Caveats and Best Practices:

- Readability: Best used for simple operations. For complex functions, define a regular function.

- One-Liners: Ideal for operations that are not reused elsewhere.

**ZIP FUNCTION:**

The `zip()` function in Python is a built-in function used to aggregate elements from two or more iterables (lists, tuples, etc.). It creates a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, then the second item, and so on.

Syntax:

zip(\*iterables)

- `\*iterables`: Two or more iterables (e.g., lists, tuples). The asterisk (\*) indicates that the function can take any number of iterables.

The resulting iterator stops when the shortest input iterable is exhausted. The items are aggregated based on their index. If the iterables are of uneven length, missing values are ignored.

Simple Example:

Suppose you have two lists and you want to pair their elements:

list1 = [1, 2, 3]

list2 = ['a', 'b', 'c']

zipped = zip(list1, list2)

print(list(zipped)) # Output: [(1, 'a'), (2, 'b'), (3, 'c')]

Advanced Use:

`zip()` can be combined with other functions like `map()` to perform complex data manipulations.

Combining Lists into a Dictionary:

list1 = [1, 2, 3]

list2 = ['a', 'b', 'c']

paired\_dict = dict(zip(list1, list2))

print(paired\_dict) # Output: {1: 'a', 2: 'b', 3: 'c'}

Unzipping Values:

The `\*` operator can be used in conjunction with `zip()` to unzip the values.

zipped\_list = [(1, 'a'), (2, 'b'), (3, 'c')]

numbers, letters = zip(\*zipped\_list)

print(numbers) # Output: (1, 2, 3)

print(letters) # Output: ('a', 'b', 'c')

Notes:

- The `zip()` function returns an iterator of tuples, which can be converted to other formats like lists or dictionaries.

- It is handy for parallel iteration over multiple iterables.

- If the input iterables are empty, `zip()` returns an empty iterator.

The `zip()` function is widely used for combining data, iterating over multiple sequences simultaneously, and creating complex data structures with ease and efficiency.

Practical Examples of Zip in Data Preparation and Business Data Scenarios

1. Merging Two Lists into a pandas DataFrame

- Combine product names and prices into a DataFrame.

```python

import pandas as pd

product\_names = ['Product A', 'Product B', 'Product C']

prices = [100, 150, 200]

df = pd.DataFrame(list(zip(product\_names, prices)), columns=['Product', 'Price'])

print(df)

```

2. Creating a Dictionary from Two Lists for Easy Lookup

- Useful for creating lookup dictionaries in business data.

```python

product\_ids = [101, 102, 103]

product\_names = ['Product A', 'Product B', 'Product C']

product\_dict = dict(zip(product\_ids, product\_names))

print(product\_dict)

```

3. Combining Multiple Arrays for Calculations

- Performing operations on corresponding elements of arrays (like NumPy arrays).

```python

import numpy as np

sales = np.array([100, 200, 300])

costs = np.array([80, 150, 240])

profit = [sale - cost for sale, cost in zip(sales, costs)]

print(profit)

```

4. Aggregating Data from Multiple DataFrames

- Combine data about the same entities from different DataFrames.

```python

df1 = pd.DataFrame({'Product\_ID': [101, 102, 103], 'Price': [10, 20, 30]})

df2 = pd.DataFrame({'Product\_ID': [101, 102, 103], 'Quantity': [3, 4, 5]})

merged\_data = pd.DataFrame(list(zip(df1['Product\_ID'], df1['Price'], df2['Quantity'])),

columns=['Product\_ID', 'Price', 'Quantity'])

print(merged\_data)

```

5. Pairing Columns from a DataFrame

- Pair up columns for further analysis or processing.

```python

df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

for a, b in zip(df['A'], df['B']):

print(f"Pair: ({a}, {b})")

```

Conclusion:

The `zip()` function is versatile and beneficial for aligning elements from different iterables. It's especially useful in data preparation and analysis, allowing for efficient manipulation, combination, and transformation of data in pandas DataFrames and Python data structures.

**Enumerate Function:**

The `enumerate()` function in Python adds a counter to an iterable and returns it in a form of an enumerate object. This object can be used directly in loops or converted into a list of tuples using list() method.

Syntax:

enumerate(iterable, start=0)

- `iterable`: Any Python iterable such as list, tuple, set, etc.

- `start` (optional): The index at which the counter starts, default is 0.

Enumerate is often used when you need a counter with the items of an iterable during a loop. It's more Pythonic and readable compared to using a manual counter with `for` loops.

Simple Example:

Consider a list of items, and you want to get the index of each item in the loop:

fruits = ['apple', 'banana', 'mango']

for index, fruit in enumerate(fruits):

print(f"{index}: {fruit}")

# Output:

# 0: apple

# 1: banana

# 2: mango

Advanced Use:

Enumerate is particularly useful with dictionaries, where items can be paired with their indices, or when indexing is essential in loops.

Custom Start Index:

You can start the counting from a number other than 0.

fruits = ['apple', 'banana', 'mango']

for index, fruit in enumerate(fruits, start=1):

print(f"{index}: {fruit}")

# Output:

# 1: apple

# 2: banana

# 3: mango

Notes:

- `enumerate()` is useful for obtaining an indexed list, where each element of a list is paired with an index.

- It enhances readability and maintainability of loops where both the elements and their indices are needed.

- The function can be combined with other Python functions and structures for more complex iterations.

The `enumerate()` function is a convenient and idiomatic way to loop through an iterable when both the element and its index are needed, making code more readable and Pythonic.