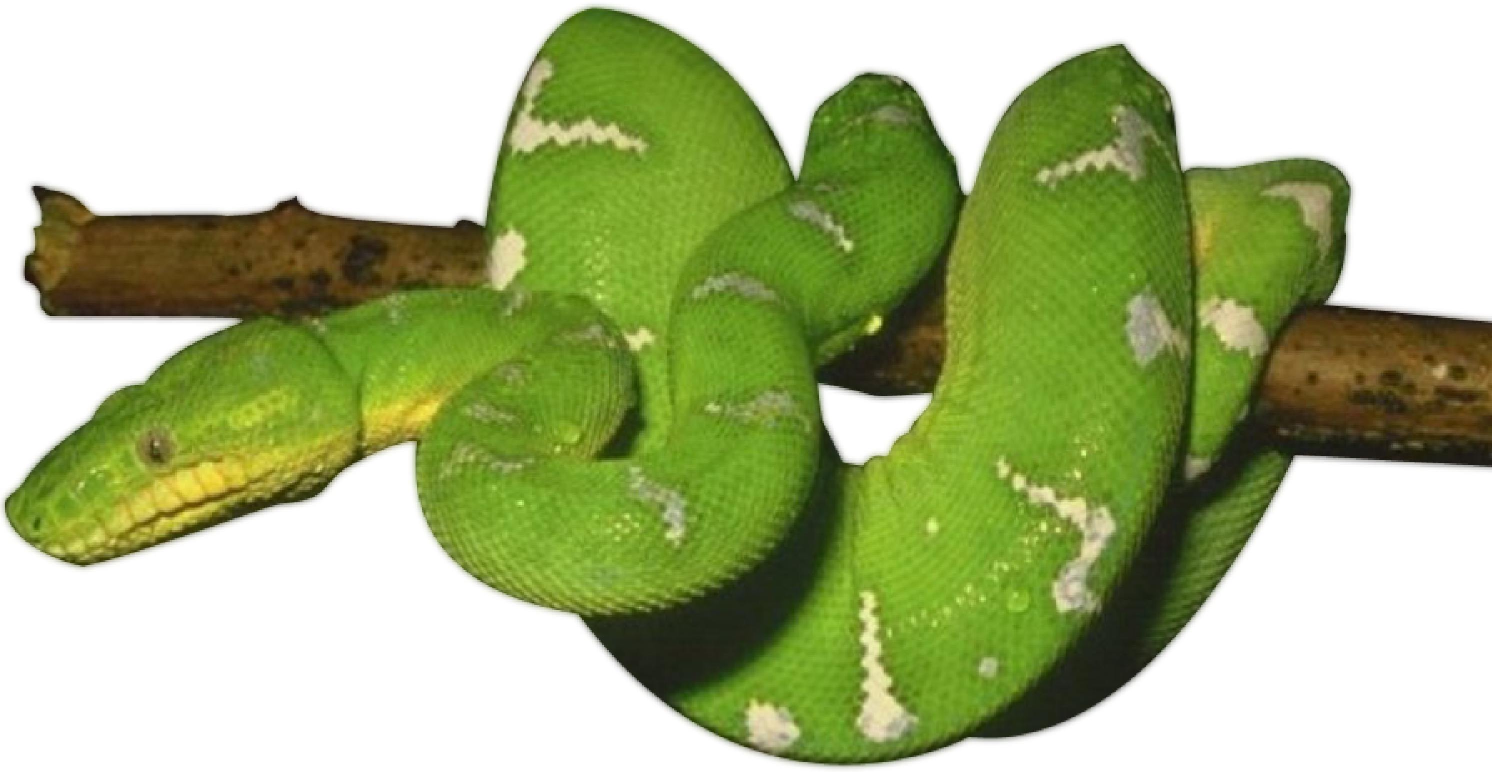
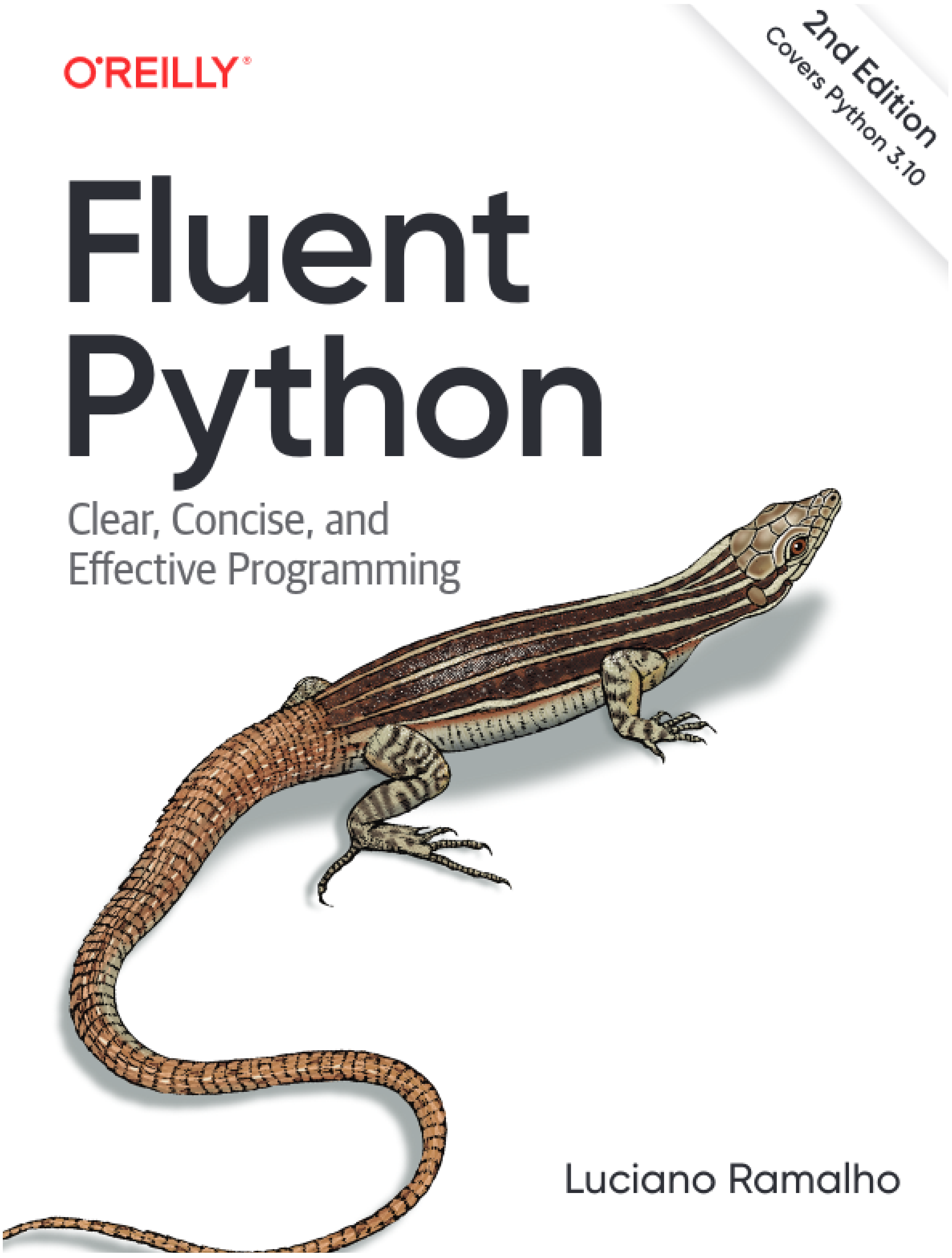
AN ARRAY OF SEQUENCES

ARIHARASUDHAN



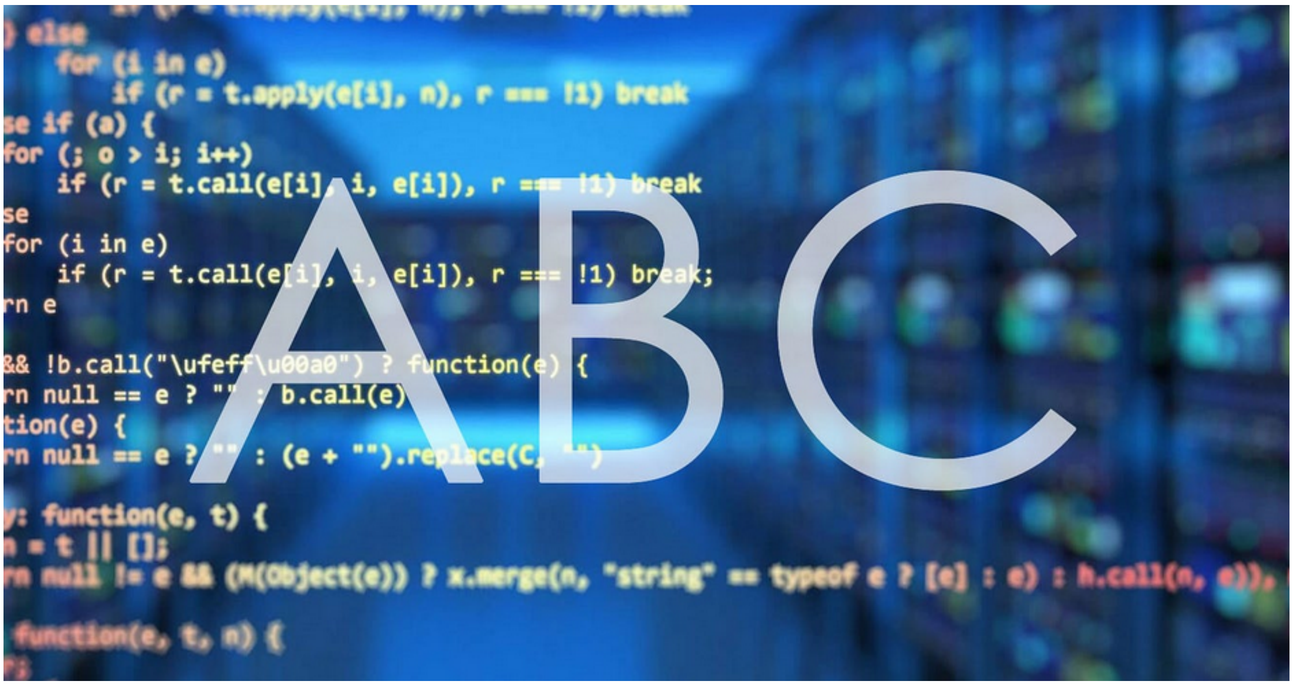
What is this short book?

This short book series is a summarization of chapters in the so called “Fluent Python” book by Luciano Ramalho.



Python inherites ABC

Python acquires most of it’s “Pythonic Features” from the language “ABC”. Guido, The Creator of Python Language contributed to ABC, A 10 years research project.



Built-In Sequences

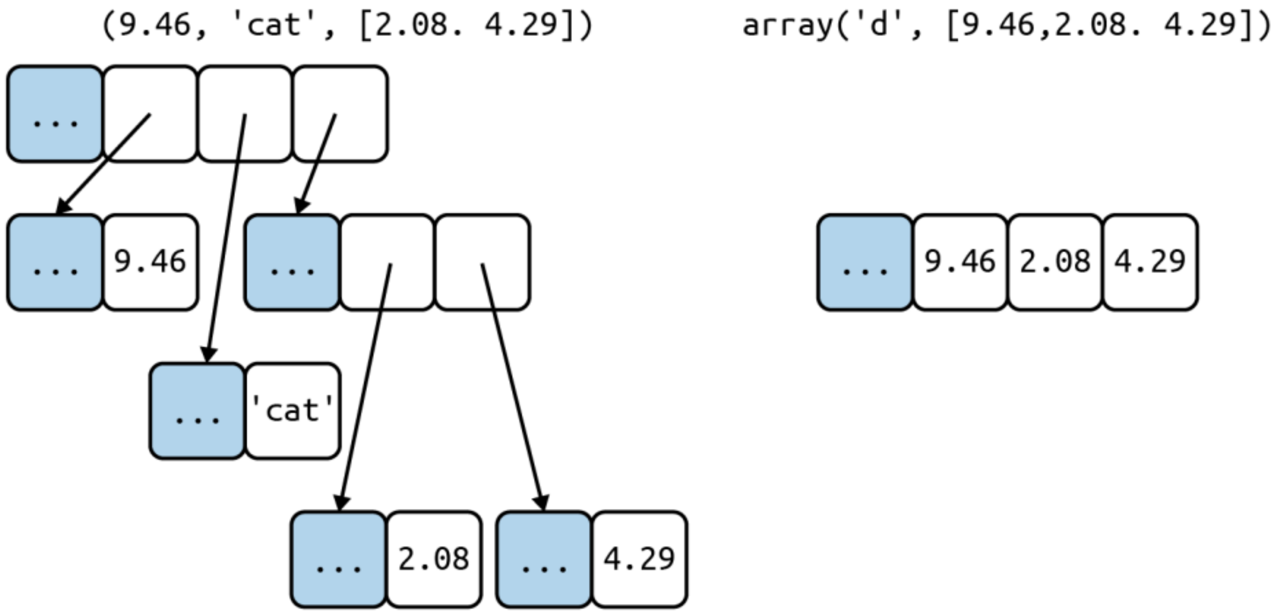
There are more sequences, implemented in C. The general two types are,

> Container Sequence

It holds “references” to items of different types.

Eg: List, Tuple

> Flat Sequence
It holds the elements of same type in it’s own memory space.
Eg: String, Bytes, Array



These are the memory diagrams for a tuple and an array, each with three items. Gray cells represent the in-memory header of each Python object. The tuple has an array of references to it’s items. Each item is a separate Python object, possibly holding references to other Python objects, like that two-item list. In contrast, the Python array is a single object, holding a C language array of three doubles.

Im-Mutable

The Another way of grouping sequence types is by mutability.
Mutable sequences: List, Bytearray, array.array, collections.deque.
Immutable sequences: Tuple, String
The common traits are mutable versus immutable; container versus flat. They are helpful to extrapolate what we know about one sequence type to others. The most fundamental sequence type is the list: a mutable container.

ListComps

A quick way to build a sequence is using a list comprehension (if the target is a list) or a generator expression (for other kinds of sequences).
Following code creates, a list of squares of numbers ranging from 0 to 9 (inc).

```
squares = [x**2 for x in range(10)]
```

We can also have a condition.

```
even_squares = [x**2 for x in range(10)
if x % 2 == 0]
```

Tip: We can write multiline listcomps as the line break is not there within [], {}, and ().

Scope in ListComps

List comprehensions, generator expressions (yet to see), and their equivalents for sets and dictionaries create variables with a local scope, limited to the expression or comprehension itself. However, variables assigned using the Walrus operator (:=) persist beyond the comprehension's scope.

```
# Without the Walrus operator
squares = [x**2 for x in range(5)]
print(squares)  # Output: [0, 1, 4, 9, 16]

# With the Walrus operator
squares = [y := x**2 for x in range(5)]
print(squares)  # Output: [0, 1, 4, 9, 16]
print(y)  # Output: 16
# print(x) causes error!
```

Map & Filter vs ListComps

ListComps offer an alternative to using map and filter functions with lambda expressions. They provide a concise syntax.

```
# Example: Squaring numbers greater
# than 5 using list comprehension
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
res = [num**2 for num in nums if num > 5]
print(res)  # Output: [36, 49, 64, 81, 100]
```

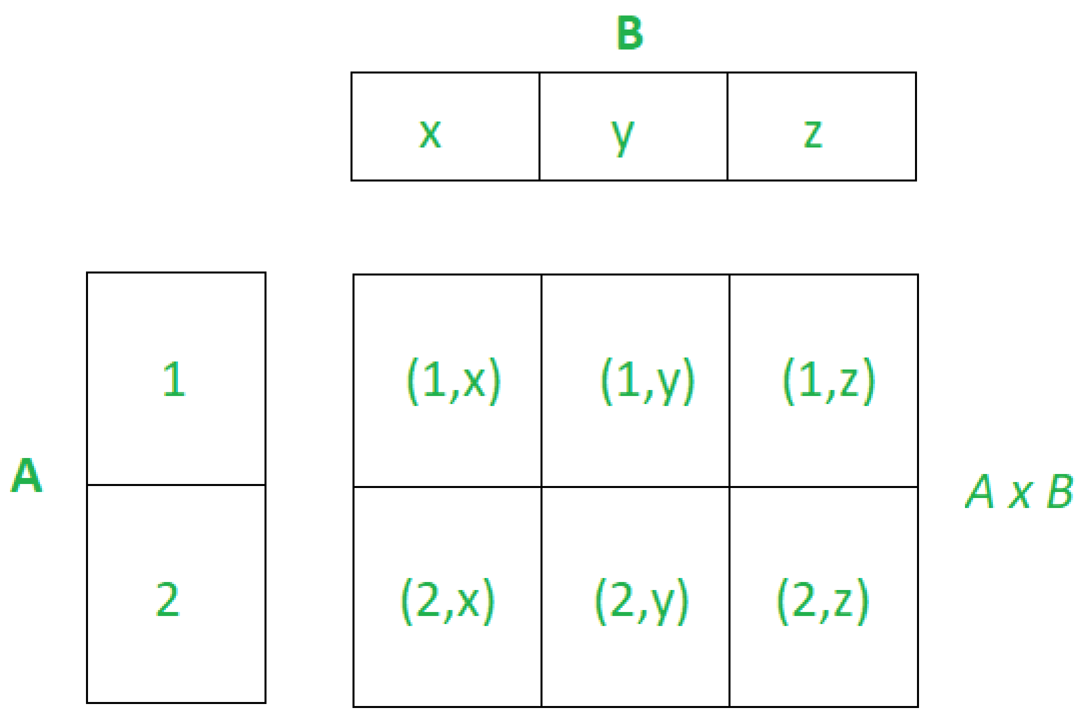
This example demonstrates how a list comprehension can be used to efficiently square numbers greater than 5 without the need for lambda functions or separate filtering steps.

Now, if it is map and filter, how can we do this?

```
res = list(map(lambda x: x**2,
filter(lambda x: x > 5, numbers)))
print(res)  # Output: [36, 49, 64, 81, 100]
```

Cartesian Product

The Cartesian product is a mathematical operation that returns a set containing all possible ordered pairs of elements from two sets.



We can achieve this using ListComps.

```
set_A = {1, 2}
set_B = {'a', 'b', 'c'}
cartesian_product = [(a, b) for a in set_A
for b in set_B]
print(cartesian_product)
# Outputs [(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'),
(2, 'b'), (2, 'c')]
```

To achieve the same using loops,

```
cartesian_product = []
for a in set_A:
    for b in set_B:
        cartesian_product.append((a, b))
print(cartesian_product)
```

GenExps

Genexps (Generator Expressions) use the same syntax as listcomps, but are enclosed in parentheses rather than brackets. A genexp saves memory because it yields items one by one using the iterator protocol instead of building a whole list.

We can initialize a tuple or an array from a generator expression like this:

```
symbols = '$¢£¥€¤'
tuple_of_symbols = tuple(ord(symbol) for symbol in symbols)
print(tuple_of_symbols)
# Output: (36, 162, 163, 165, 8364, 164)
```

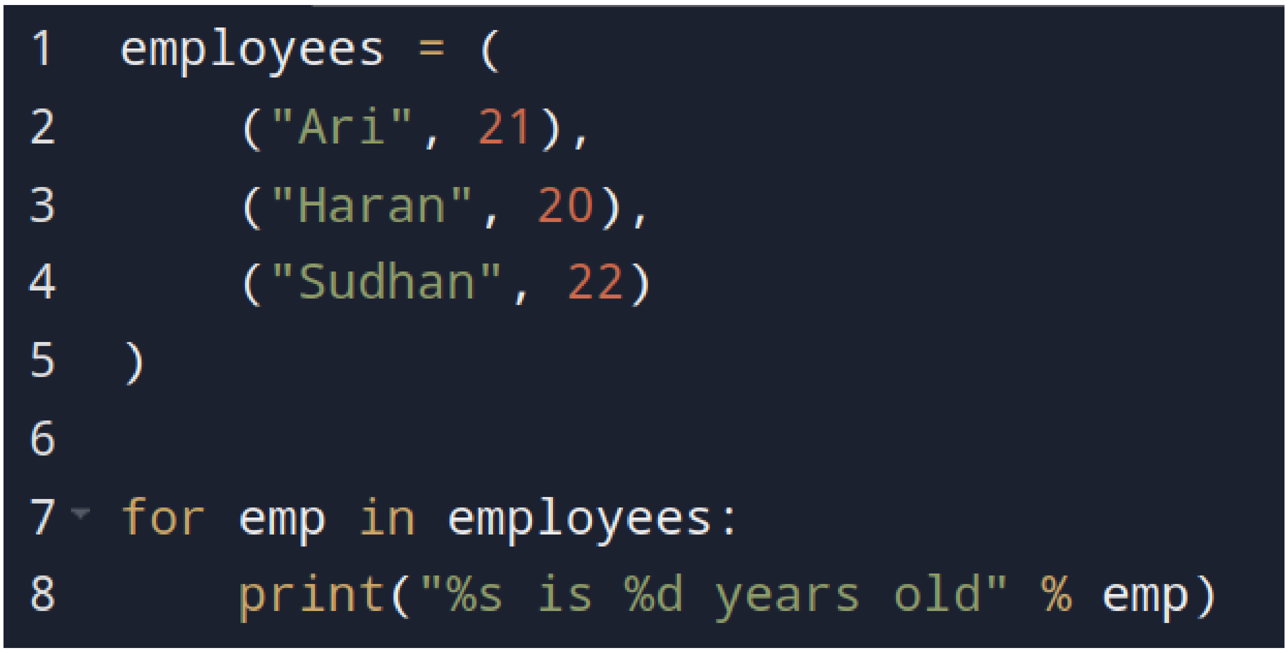
Generator expressions are particularly useful when dealing with large sequences or when memory efficiency is important. For example, we can use a generator expression to compute a Cartesian product without building the entire list in memory:

```
colors = ['black', 'white']
sizes = ['S', 'M', 'L']
for tshirt in (f'{c} {s}' for c in colors for s
in sizes):
    print(tshirt)
```

This example yields T-shirt variations one by one without building the entire list, saving memory especially if the input lists are large. Overall, generator expressions are versatile tools for initializing sequences or producing output without the need to keep everything in memory at once.

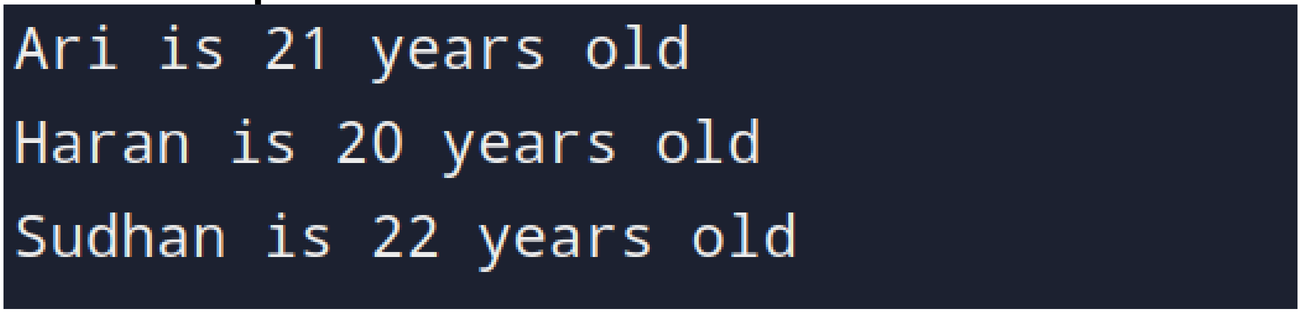
Tuples

Python is generally defined as “A List that is immutable (not changeable)”. Another overlooked use of tuple is using it as a container of records.



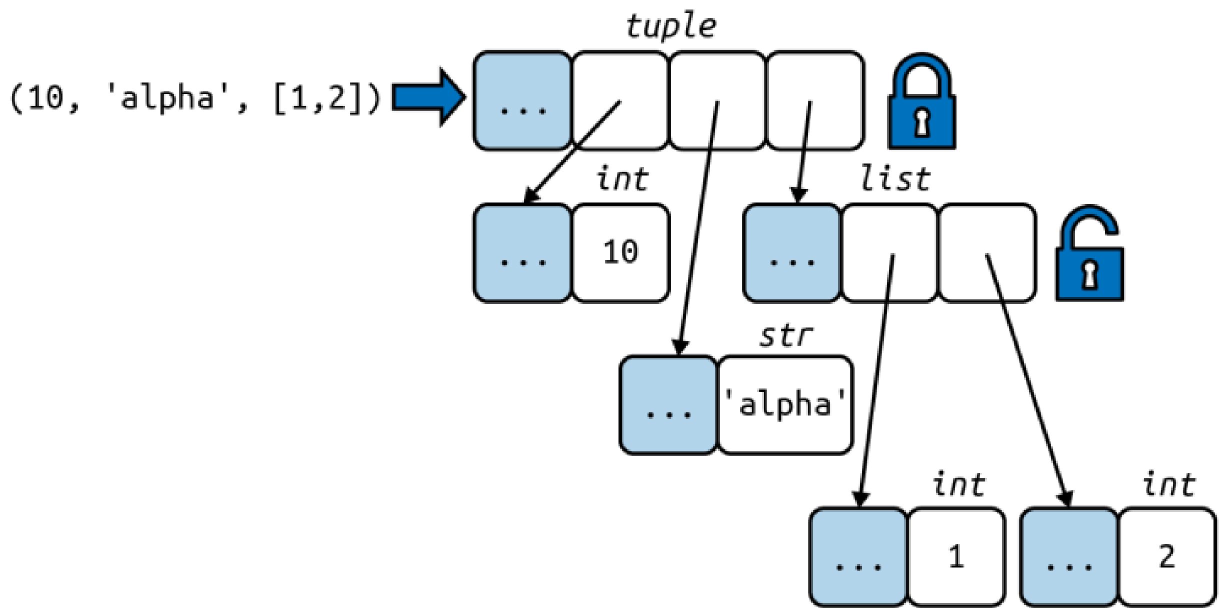
```
employees = (
    ("Ari", 21),
    ("Haran", 20),
    ("Sudhan", 22)
)

for emp in employees:
    print("%s is %d years old" % emp)
```

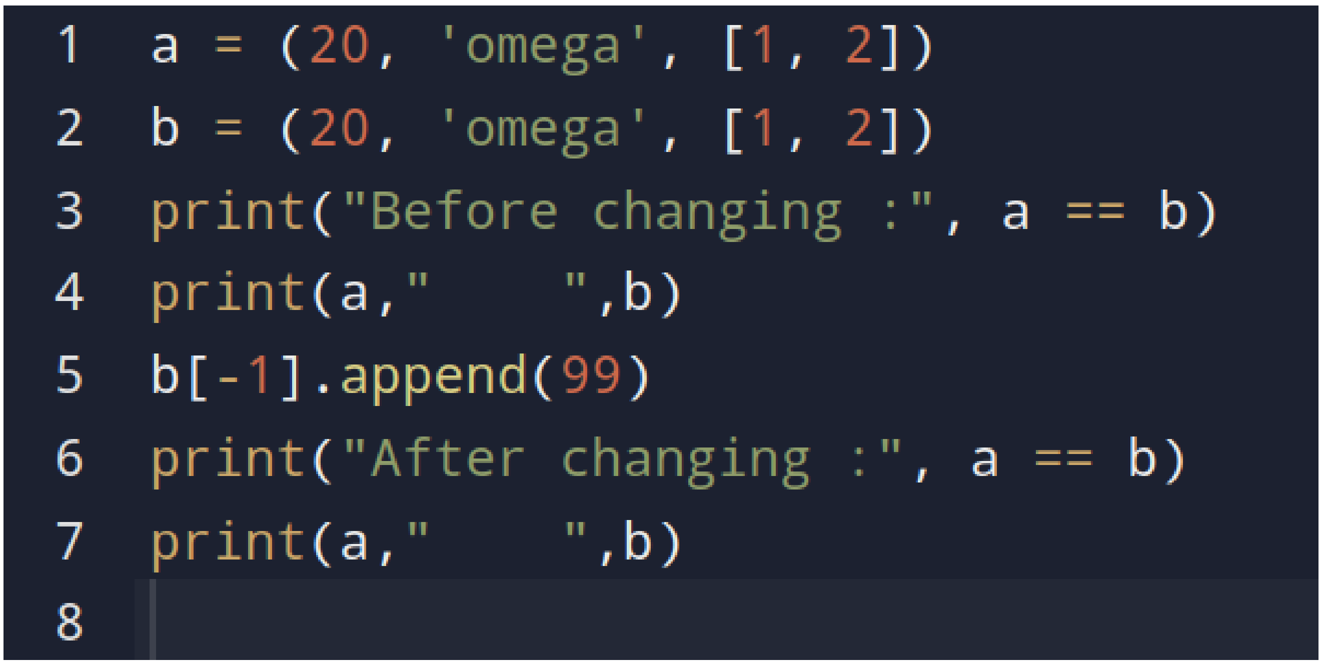
The output would be:



```
Ari is 21 years old
Haran is 20 years old
Sudhan is 22 years old
```

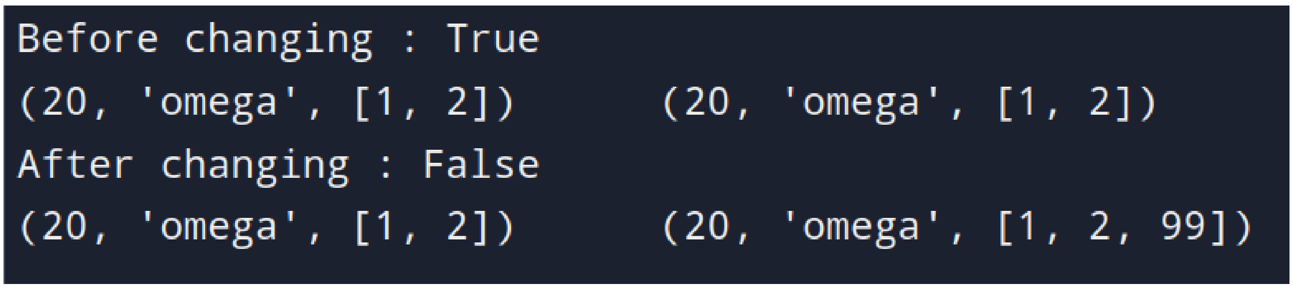
If a tuple element is mutable, that element can be changed. It implies that the immutability of a tuple only applies to the references contained in it. References in a tuple cannot be deleted or replaced. But if one of those references points to a mutable object, and that object is changed, then the value of the tuple changes.





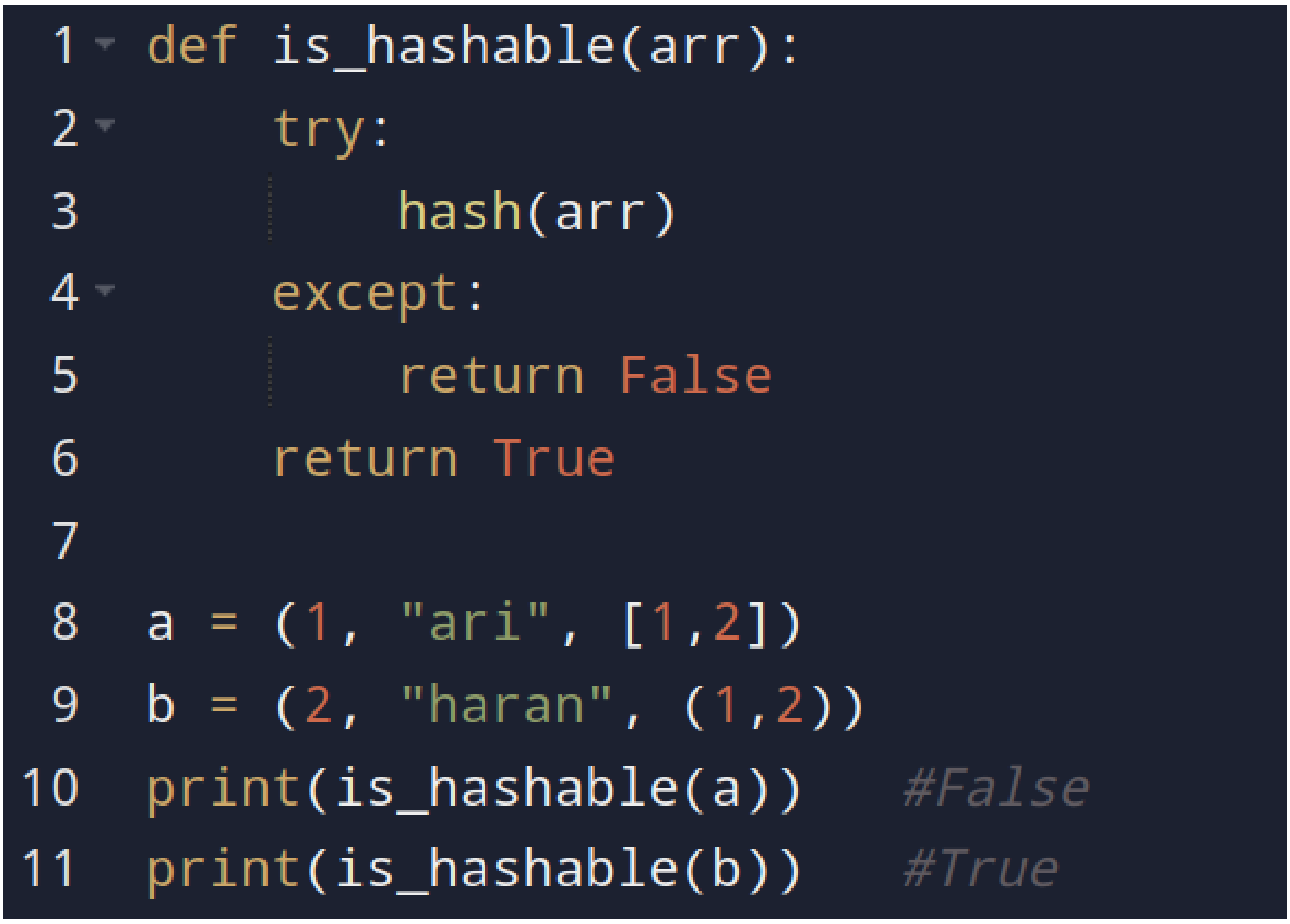
```python
a = (20, 'omega', [1, 2])
b = (20, 'omega', [1, 2])
print("Before changing :", a == b)
print(a,"     ",b)
b[-1].append(99)
print("After changing :", a == b)
print(a,"     ",b)
```

In the example given above, when a mutable object is changed, the reference also changes.



```
Before changing : True
(20, 'omega', [1, 2])      (20, 'omega', [1, 2])
After changing : False
(20, 'omega', [1, 2])      (20, 'omega', [1, 2, 99])
```

An object is only hashable only when it has immutable elements. If a tuple has a immutable data, it can’t be used as a key (NOT HASHABLE!). We can use the hash() function in Python to check whether a given tuple is hashable or not.



```python
def is_hashable(arr):
    try:
        hash(arr)
    except:
        return False
    return True

a = (1, "ari", [1,2])
b = (2, "haran", (1,2))
print(is_hashable(a))   #False
print(is_hashable(b))   #True
```

Unpacking Sequences

To avoid using error-prone, unwanted indices to get elements and set them to variables, we can use unpacking instead.

```python
record = ("ari", 21, 6382509390)
name, age, num = record
print(f'{name.title()} is {age} years old and can be contacted using +91 {num}')
```

Here, the variables, name, age and num get their values from the unpacked record. When calling a function, we can unpack a record by prefixing *.

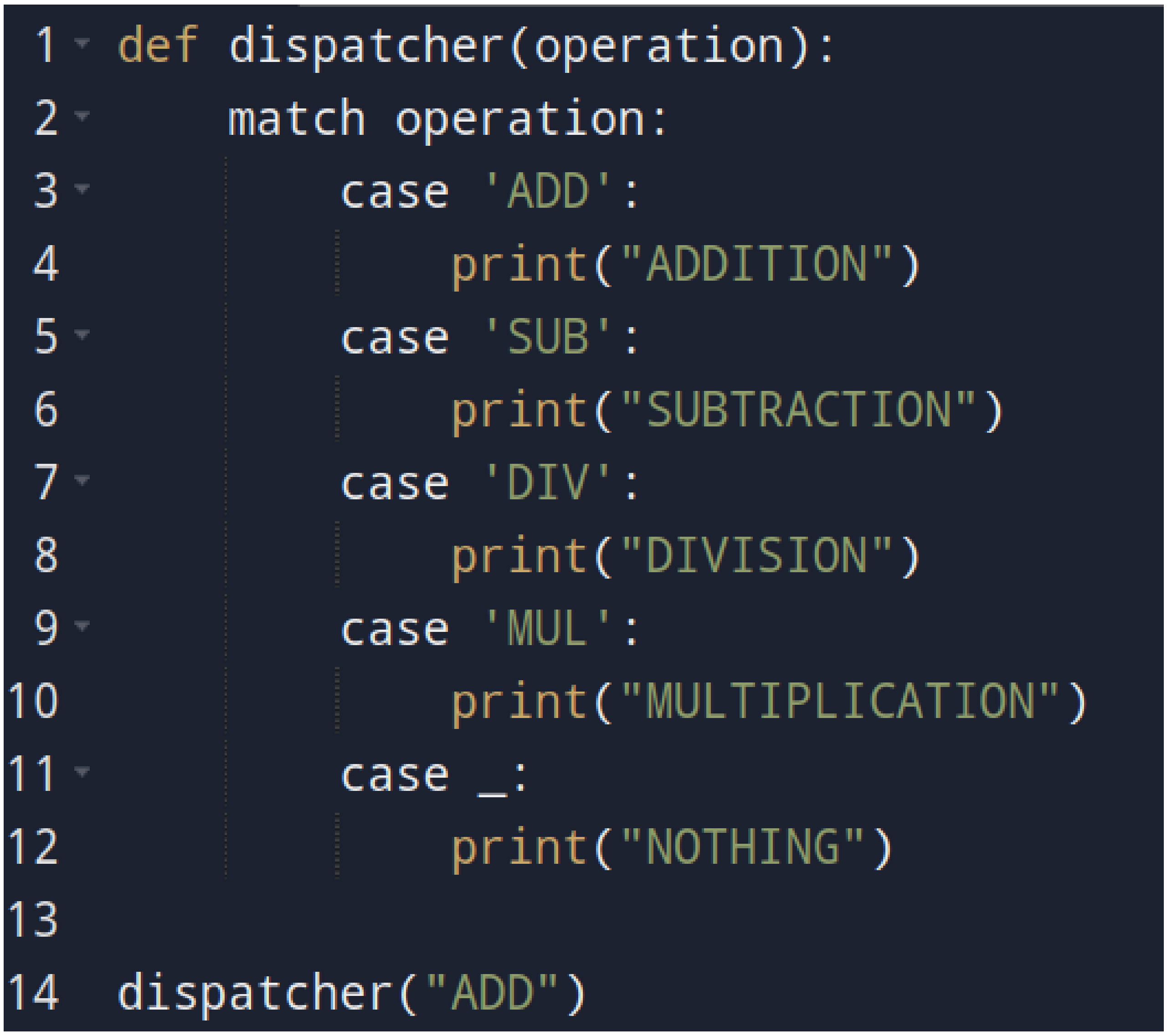
```python
def show_record(name, age):
    print(f"{name} is {age} years old!")

record = ("Ari", 21)
show_record(*record)  #prints "Ari is 21 years old!"
```

We can also grab excess items using the following ways:

```python
LIST = list(range(10))
a,b, *c, d,e = LIST
print(a,b,c,d,e) # 0 1 [2, 3, 4, 5, 6, 7] 8 9

a,b,c, *d = LIST
print(a,b,c,d) # 0 1 2 [3, 4, 5, 6, 7, 8, 9]

*a, b = LIST
print(a,b)      # [0, 1, 2, 3, 4, 5, 6, 7, 8] 9
```

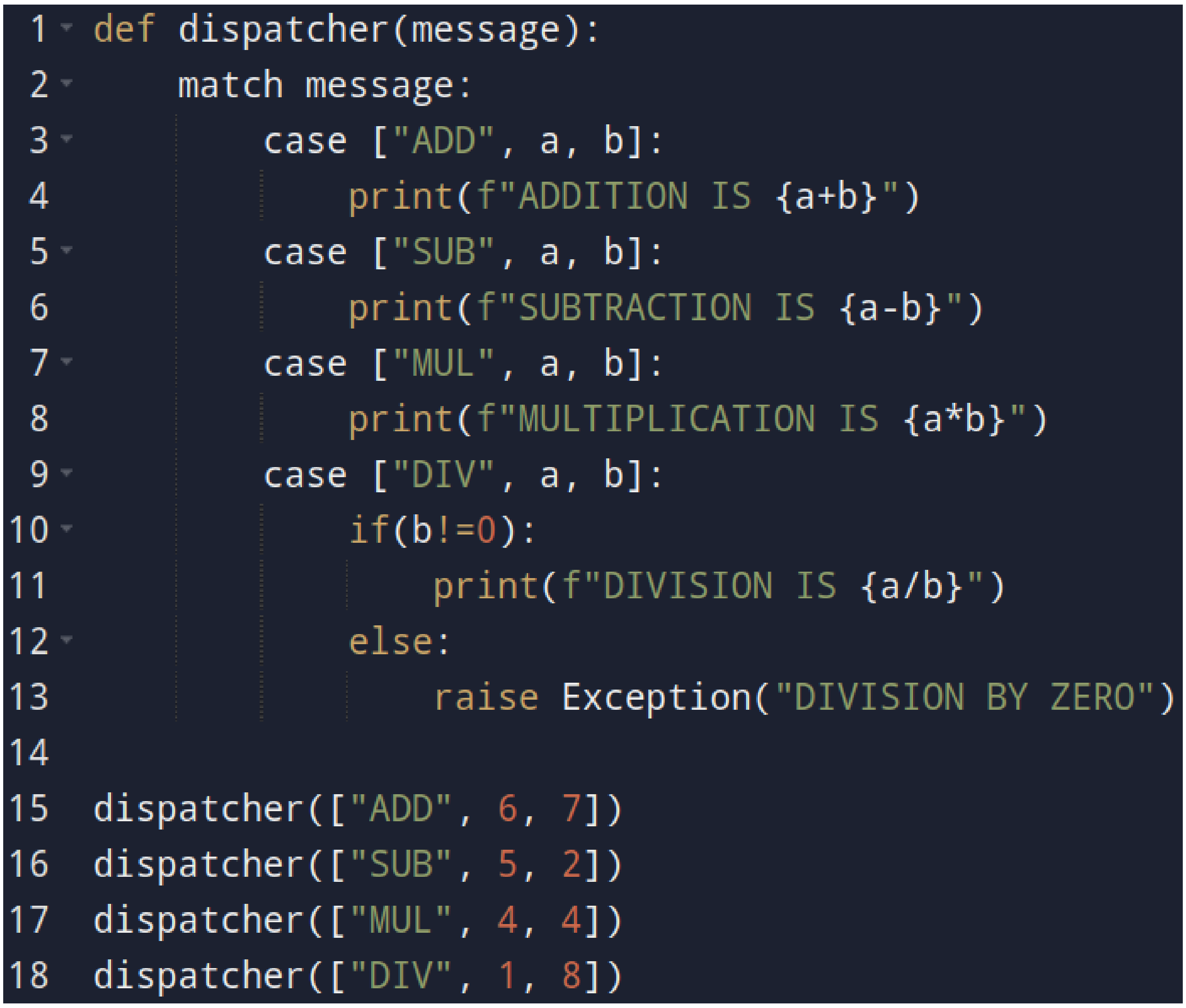
Pattern Matching with Seqs

I hope you guys know of SWITCH-CASE stuffs in other languages. Python has introduced the equivalent, match/case statement.



```python
def dispatcher(operation):
    match operation:
        case 'ADD':
            print("ADDITION")
        case 'SUB':
            print("SUBTRACTION")
        case 'DIV':
            print("DIVISION")
        case 'MUL':
            print("MULTIPLICATION")
        case _:
            print("NOTHING")

dispatcher("ADD")
```

The above example code will output “ADDITION”. The expression after the keyword match is Subject. Another example is as following:

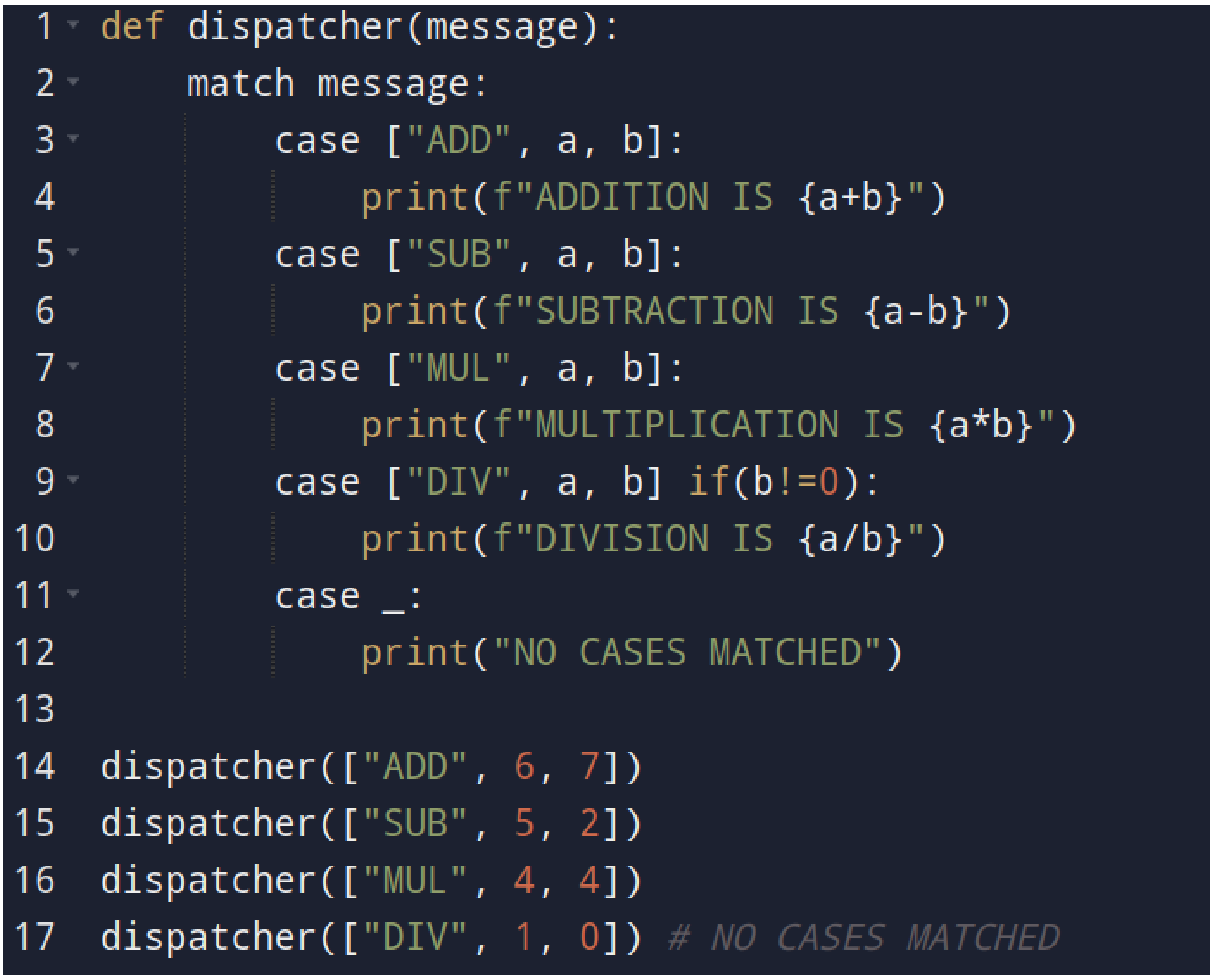


```python
def dispatcher(message):
    match message:
        case ["ADD", a, b]:
            print(f"ADDITION IS {a+b}")
        case ["SUB", a, b]:
            print(f"SUBTRACTION IS {a-b}")
        case ["MUL", a, b]:
            print(f"MULTIPLICATION IS {a*b}")
        case ["DIV", a, b]:
            if(b!=0):
                print(f"DIVISION IS {a/b}")
            else:
                raise Exception("DIVISION BY ZERO")

dispatcher(["ADD", 6, 7])
dispatcher(["SUB", 5, 2])
dispatcher(["MUL", 4, 4])
dispatcher(["DIV", 1, 8])
```

It will output,

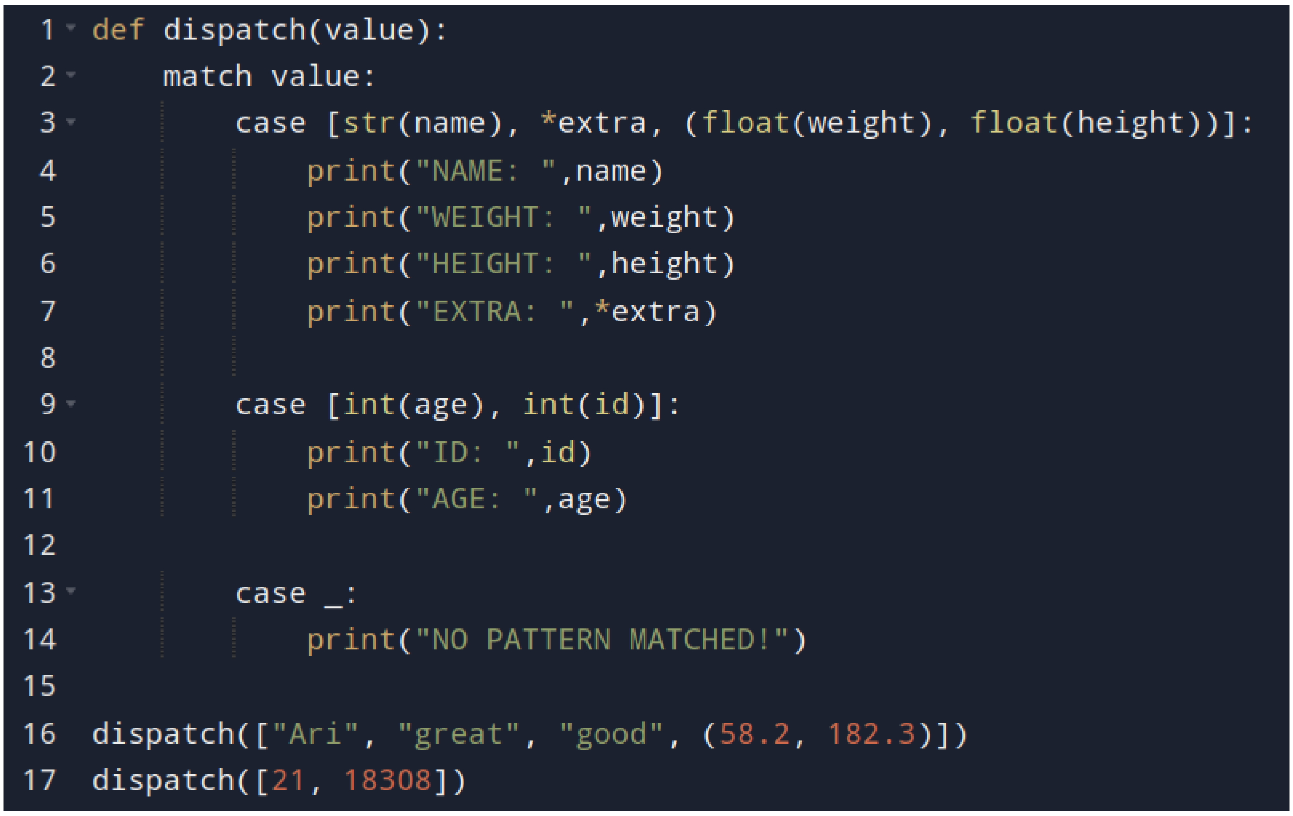
```
ADDITION IS 13
SUBTRACTION IS 3
MULTIPLICATION IS 16
DIVISION IS 0.125
```

You can look into the case of DIV where we use a if condition to check whehter the second value is zero or not. We can make it simple by writing a if right after the case statement as shown below.



```python
def dispatcher(message):
    match message:
        case ["ADD", a, b]:
            print(f"ADDITION IS {a+b}")
        case ["SUB", a, b]:
            print(f"SUBTRACTION IS {a-b}")
        case ["MUL", a, b]:
            print(f"MULTIPLICATION IS {a*b}")
        case ["DIV", a, b] if(b!=0):
            print(f"DIVISION IS {a/b}")
        case _:
            print("NO CASES MATCHED")

dispatcher(["ADD", 6, 7])
dispatcher(["SUB", 5, 2])
dispatcher(["MUL", 4, 4])
dispatcher(["DIV", 1, 0]) # NO CASES MATCHED
```

Watch out the if condition after the case of DIV. We can match patterns actually... It means, rather than just providing a data to the match, we can follow a pattern in cases. Only the matched case will be executed. Look into the following example for how this pattern matching actually works!



```python
def dispatch(value):
    match value:
        case [str(name), *extra, (float(weight), float(height))]:
            print("NAME: ",name)
            print("WEIGHT: ",weight)
            print("HEIGHT: ",height)
            print("EXTRA: ",*extra)

        case [int(age), int(id)]:
            print("ID: ",id)
            print("AGE: ",age)

        case _:
            print("NO PATTERN MATCHED!")

dispatch(["Ari", "great", "good", (58.2, 182.3)])
dispatch([21, 18308])
```

It outputs,

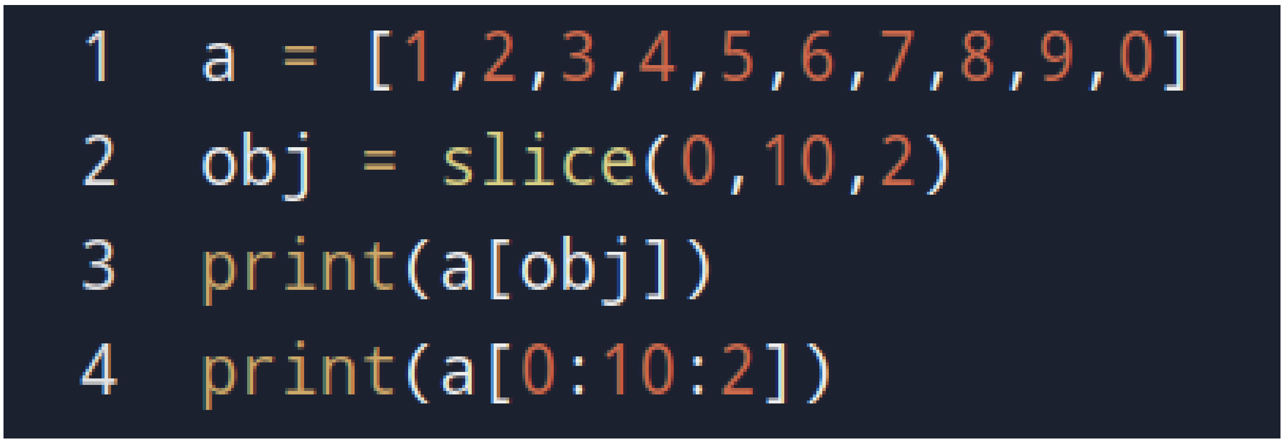
```
NAME:  Ari
WEIGHT:  58.2
HEIGHT:  182.3
EXTRA:  great good
ID:  18308
AGE:  21
```

Slicing

A common feature of list, tuple, str, and all sequence types in Python is slicing!

```
l = [10, 20, 30, 40, 50, 60]
>>> l[:2]
# Outputs [10, 20]

>>> l[2:]
# Outputs [30, 40, 50, 60]

>>> l[:3] # split at 3
# Outputs [10, 20, 30]

>>> l[3:]
# Outputs [40, 50, 60]

>>> l[::2] # 3rd value is a stride/step
# Outputs [10, 30, 50]
```

Actually, when we write a slice like l[1:2:2] or like whatever, there will be a slice object created. Instead of directly specifying the slice values such as start, end and stride, let's create a slice object by ourself and understand it.



```python
a = [1,2,3,4,5,6,7,8,9,0]
obj = slice(0,10,2)
print(a[obj])
print(a[0:10:2])
```

Both results [1, 3, 5, 7, 9]. A slice object will be created whenever we perform slicing using [start: end: stride]. The created slice object will be passed to the dunder method, __getitem__().

Slicing n-Dimensions

In the external NumPy package, items of a two dimensional numpy.ndarray can be fetched using the syntax a[i, j]. A two-dimensional slice can be obtained with an expression like a[m:n, k:l]. Ellipsis can also be used to slice Multi Dimensional Arrays. If x is a four-dimensional array, x[i, ...] is a shortcut for x[i, :, :, :,].

Slices can be assigned!

```
l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> l[2:5] = [20, 30]
>>> l
[0, 1, 20, 30, 5, 6, 7, 8, 9]

>>> del l[5:7]
>>> l
[0, 1, 20, 30, 5, 8, 9]

>>> l[3::2] = [11, 22]
>>> l
[0, 1, 20, 11, 5, 22, 9]

>>> l[2:5] = 100
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: can only assign an iterable

>>> l[2:5] = [100]
>>> l
[0, 1, 100, 22, 9]
```

Using + and * with Sequences

Both operands of + must be of the same sequence type, and neither of them is modified, but a new sequence of that same type is created as result of the concatenation.

```
>>> l = [1, 2, 3]
>>> l * 5
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 5 * 'abcd'
'abcdabcdabcdabcdabcd'
```

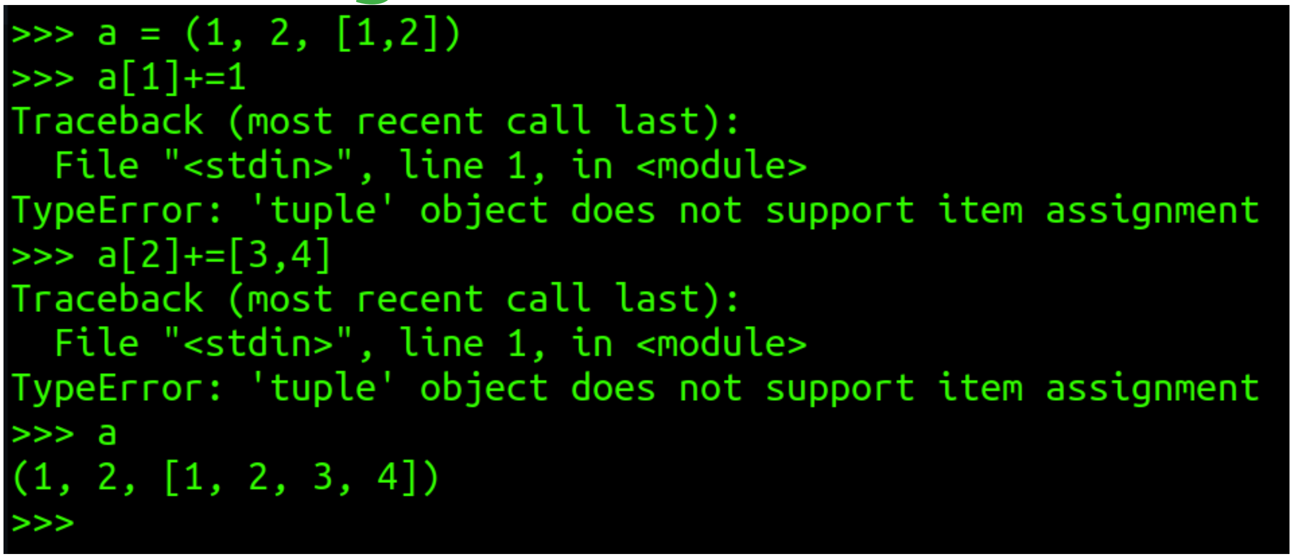
If you multiply a collection with a magnitude n, it will result in an array of n elements for each elements in the original array.

```
>>> board = [['_'] * 3 for i in range(3)]
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[1][2] = 'X'
>>> board
[['_', '_', '_'], ['_', '_', 'X'], ['_', '_', '_']]
```

+ can be used for concatenating two collections.

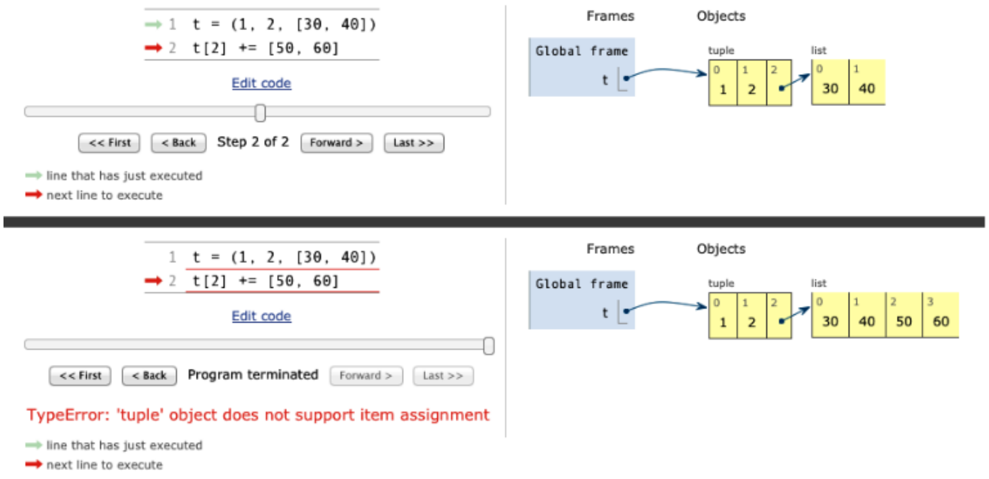
```
a = [1,2] + [3,4]          # a has [1,2,3,4]
b =  “Hi “ + “Hello”       # b has “Hi Hello”
```

A+=Assignment Puzzler



```
>>> a = (1, 2, [1,2])
>>> a[1]+=1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> a[2]+=[3,4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> a
(1, 2, [1, 2, 3, 4])
>>>
```

Usually, [1,2] + [3,4] will result in, [1,2,3,4]. But, if the array is in a tuple and if we try to do the same, it's a weird stuff!!! Look at the above image, where we try to change element in 1st index and get an error. But, when we tried to add a list to the list in 2nd index, it threw an error but resulted an expected output! What a puzzle!

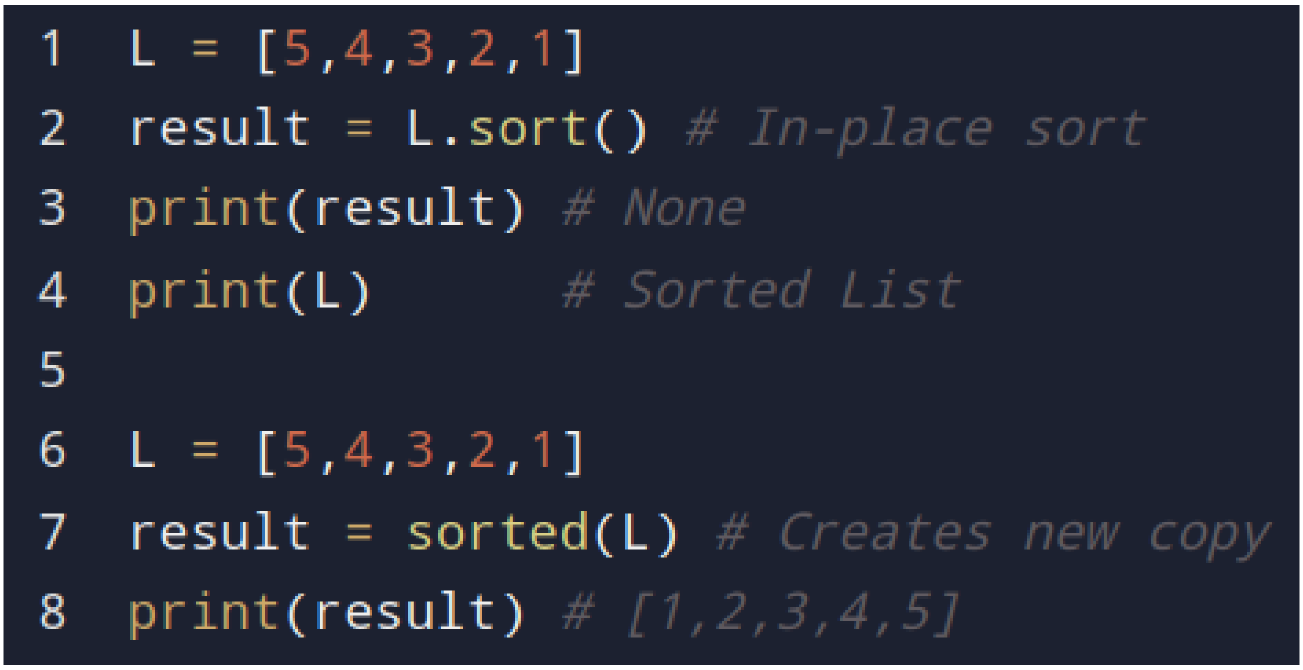


```
t = (1, 2, [30, 40])
t[2] += [50, 60]
```

TypeError: 'tuple' object does not support item assignment

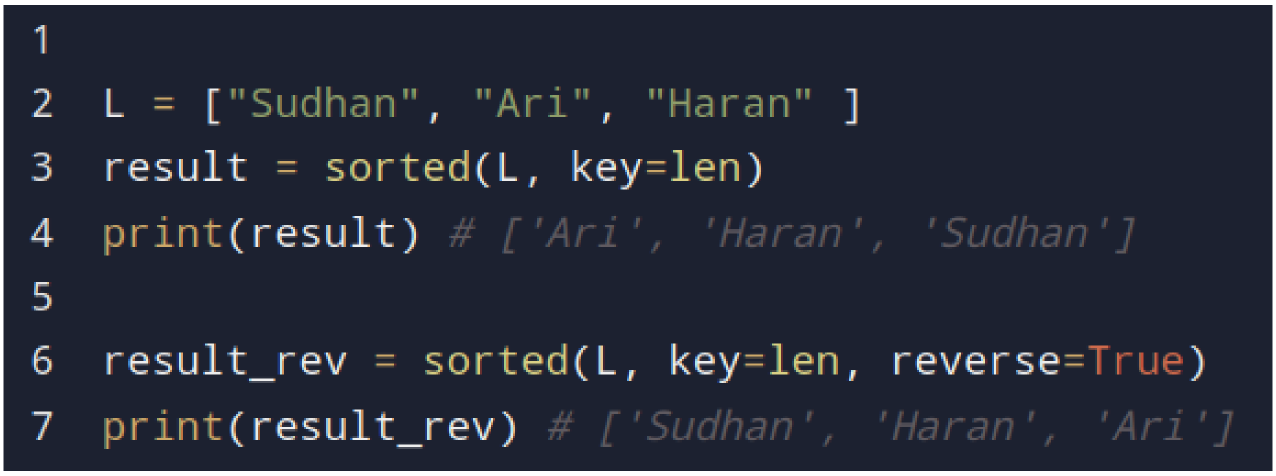
list.sort() vs sorted()

The list.sort() sorts a list in place and returns None. It doesn't make a copy of the list. The returned None indicates that there is no new object created. Meanwhile, the built-in function sorted creates a new list and returns it. It accepts any iterable object as an argument, including immutable sequences and generators.



```python
L = [5,4,3,2,1]
result = L.sort() # In-place sort
print(result) # None
print(L)         # Sorted List

L = [5,4,3,2,1]
result = sorted(L) # Creates new copy
print(result) # [1,2,3,4,5]
```

Both list.sort() and sorted() takes two optional keywords. (1) reverse: If True, the items are returned in descending order. (2) key: A one-argument function that will be applied to each item to produce its sorting key. For example, when sorting a list of strings, key=str.lower can be used to perform a case-insensitive sort, and key=len will sort the strings by character length. Let’s talk of them now in the following example.



```python

L = ["Sudhan", "Ari", "Haran" ]
result = sorted(L, key=len)
print(result) # ['Ari', 'Haran', 'Sudhan']

result_rev = sorted(L, key=len, reverse=True)
print(result_rev) # ['Sudhan', 'Haran', 'Ari']
```

Beyond Lists

The list type is flexible and easy to use, but depending on specific requirements, there are better options. For example, an array saves a lot of memory when you need to handle millions of floating-point values. On the other hand, if you are constantly adding and removing items from opposite ends of a list, it’s good to know that a deque (double-ended queue) is a more efficient data structure.

Arrays

Arrays in Python offer a more memory-efficient alternative to lists, especially when dealing with sequences of numbers. They support all mutable sequence operations, such as .pop, .insert, and .extend, along with additional methods for fast loading and saving like .frombytes and .tofile.

When you create an array, you specify a typecode, indicating the underlying C type used to store each item. For example, 'd' represents double-precision floating-point numbers. This allows arrays to hold packed bytes representing machine values, similar to arrays in C. For instance, an array created with typecode 'b' stores each item as a single byte interpreted as an integer.

Here's an example demonstrating the creation, saving, and loading of a large array of floats:

```
from array import array
from random import random

# Create array of double-precision floats
floats = array('d', (random() for i in range(10**7)))

# Save the array to a binary file
with open('floats.bin', 'wb') as fp:
    floats.tofile(fp)

# Create an empty array of doubles
floats2 = array('d')

#Read 10 million nums from binary file
with open('floats.bin', 'rb') as fp:
    floats2.fromfile(fp, 10**7)

# Verify: contents of the arrays match
print(floats2 == floats)  # Output: True
```

The array.tofile and array.fromfile methods are efficient and easy to use. They are significantly faster than reading/writing from/to text files, especially for large datasets.

For instance, reading 10 million double-precision floats from a binary file is nearly 60 times faster than reading from a text file and involves less overhead in terms of file size.

For tasks involving binary data like raster images, Python provides the bytes and bytearray types. Arrays don't have an in-place sort method like lists, but you can use the sorted function to sort them. Additionally, to keep a sorted array sorted while adding items, you can use bisect.insort.

Memory View

The memoryview class in Python offers a way to handle slices of arrays without copying bytes, akin to the functionality provided by the NumPy library. It allows sharing memory between different data structures, such as PIL images, SQLite databases, and NumPy arrays, without the need for duplication. This capability is particularly valuable for managing large datasets efficiently.

Similar to the array module, memoryview provides the cast method, which enables you to reinterpret bytes as different data types without altering the underlying memory. Each call to memoryview.cast returns a new memoryview object that shares the same memory as the original.
Here's an example demonstrating the creation of different views on the same array of 6 bytes, allowing it to be treated as a 1x6, 2x3, or 3x2 matrix:

```
from array import array
# Build an array of 6 bytes
octets = array('B', range(6))
# Create a memoryview from the array
m1 = memoryview(octets)
print(m1.tolist())  # [0,1,2,3,4, 5]
#Create memoryview with a 2x3 view
m2 = m1.cast('B', [2, 3])
print(m2.tolist()) #[[0,1,2], [3,4,5]]
# Create memoryview with a 3x2 view
m3 = m1.cast('B', [3, 2])
print(m3.tolist()) #[[0, 1], [2, 3], [4, 5]]
```

```
# Modify elements in m2 and m3,
# reflecting changes in the original array
m2[1, 1] = 22
m3[1, 1] = 33
print(octets)
#array('B', [0, 1, 2, 33, 22, 5])
```

The power of memoryview can also be used to manipulate individual bytes within array elements. Here's an example of changing the value of a 16-bit integer array item by modifying one of its bytes:

```
import array
# Array of 5 16-bit signed integers
numbers = array.array('h', [-2, -1, 0, 1, 2])
# Create a memoryview from the array
memv = memoryview(numbers)
# Cast the elements of memv to bytes
memv_oct = memv.cast('B')
print(memv_oct.tolist())
# [254, 255, 255, 255, 0, 0, 1, 0, 2, 0]
# Change the value of byte offset 5
memv_oct[5] = 4
```

```
print(numbers)
# Output: array('h', [-2, -1, 1024, 1, 2])
```

DeQueues

Deques, short for double-ended queues, offer fast insertion and removal from both ends compared to lists. While lists can serve as stacks or queues, deque is more efficient for such operations, especially when inserting or removing from the beginning of a list, which requires shifting the entire list in memory. collections.deque is a thread-safe implementation of a double-ended queue, optimized for fast operations on both ends. It's particularly useful for scenarios where you need to maintain a list of "last seen items" or similar data, as it can be bounded, meaning it can be created with a fixed maximum length. When a bounded deque is full and a new item is added, it automatically discards an item from the opposite end.

```
from collections import deque
# Create a deque with a maximum length of 10
dq = deque(range(10), maxlen=10)
print(dq)  # Output: deque([0, 1, 2, 3, 4,
5, 6, 7, 8, 9], maxlen=10)
# Rotate the deque by 3 elements to the right
dq.rotate(3)
print(dq)  # Output: deque([7, 8, 9, 0, 1,
2, 3, 4, 5, 6], maxlen=10)

# Rotate the deque by 4 elements to the left
dq.rotate(-4)
print(dq)  # Output: deque([1, 2, 3, 4, 5,
6, 7, 8, 9, 0], maxlen=10)

# Append an element to the left
dq.appendleft(-1)
print(dq)  # Output: deque([-1, 1, 2, 3, 4,
5, 6, 7, 8, 9], maxlen=10)

# Extend the deque to the right with a list of
elements
dq.extend([11, 22, 33])
print(dq)  # Output: deque([3, 4, 5, 6, 7,
8, 9, 11, 22, 33], maxlen=10)
```

```
# Extend the deque to the left with a list of elements
dq.extendleft([10, 20, 30, 40])
print(dq)    # deque([40, 30, 20, 10, 3, 4, 5, 6, 7, 8], maxlen=10)
```

The maxlen argument sets the maximum number of items allowed in the deque, and it's a read-only attribute. Rotating with a positive value takes items from the right end and moves them to the left, while a negative value takes items from the left and moves them to the right. Deque provides methods like appendleft, popleft, and rotate which are not available in lists. However, removing items from the middle of a deque is not as efficient as appending or popping from the ends. Deques are thread-safe and can be used as FIFO queues in multithreaded applications without requiring locks. Other queue implementations in the Python standard library include queue, asyncio (for asynchronous programming), and heapq (for heap-based queues).

NumPy

NumPy is a powerhouse for scientific computing in Python, offering advanced array and matrix operations. It introduces multi-dimensional, homogeneous arrays and matrices capable of holding not only numbers but also user-defined records, facilitating efficient element-wise operations.

To demonstrate the basics of NumPy, consider the following example, which showcases operations with two-dimensional arrays:

```
import numpy as np
# Create a NumPy array with integers
from 0 to 11
a = np.arange(12)
print(a)  # Output: [ 0 1 2 3 4 5 6 7 8 9 10
11]

# Inspect the type of the array
print(type(a))      #    Output:    <class
'numpy.ndarray'>
```

```python
# Inspect the dimensions of the array
print(a.shape)  # Output: (12,)

# Reshape the array into a 3x4 matrix
a.shape = 3, 4
print(a)
# Output:
# [[ 0 1 2 3]
#  [ 4 5 6 7]
#  [ 8 9 10 11]]

# Get the row at index 2
print(a[2])  # Output: [ 8 9 10 11]
# Get the element at index 2, 1
print(a[2, 1])  # Output: 9
# Get the column at index 1
print(a[:, 1])  # Output: [1 5 9]

# Transpose the array
print(a.transpose())
# Output:
# [[ 0 4 8]
#  [ 1 5 9]
#  [ 2 6 10]
#  [ 3 7 11]]
```

In addition to basic array operations, NumPy supports high-level operations such as loading, saving, and operating on all elements of an array.

```
# Load 10 million floating-point numbers from a text file
floats = np.loadtxt('floats-10M-lines.txt')

# Inspect the last three numbers
print(floats[-3:])

# Multiply every element in the floats array by 0.5
floats *= 0.5

# Inspect the last three elements again
print(floats[-3:])

# Save the array in a .npy binary file
np.save('floats-10M', floats)

# Load the data as a memory-mapped file into another array
floats2 = np.load('floats-10M.npy', 'r+')
```

```
# Multiply every element by 6 and inspect the last three elements
print(floats2[-3:])
```

NumPy and SciPy are fundamental libraries in scientific computing, forming the basis of other powerful tools like Pandas and scikit-learn. These libraries are highly efficient, with most functions implemented in C or C++, allowing them to leverage multiple CPU cores effectively. While this overview merely scratches the surface, it underscores the significance of NumPy arrays in Python's scientific ecosystem.

MERCI