Working with Data Structures 1

What data structures do we have in Python?

- Lists
- Tuples
- Dictionaries
- Sets

Summary

- Lists
- Lambda functions
- Built-in functions when working with lists

Lists

Lists

A **list** in Python is a collection that can store multiple items in a single variable. Think of it like a container where you can keep values, such as numbers, words, or even other lists.

- Lists are ordered: Items have a fixed position (called an index)
- Lists are changeable (mutable): You can add, change, or remove items
- Lists can contain different data types: numbers, strings, boolean, etc.
- Lists are written with square brackets: [ ]

Lists

- Lists can be hybrid

```
my_list = list()                    # empty list
my_list = []                        # empty list
my_list = [1, "hello", 3.14]        # list with values
my_list = [1, ]                     # list containing [1]
my_list = [1] * 3                   # list containing [1, 1, 1]
```

Lists

- Accessing elements and list slicing

```
my_list = [1, "hello", 3.14]
my_list[0]                      # 1
my_list[-1]                     # 3.14
my_list[-2]                     # 'hello'
my_list[:2]                     # [1, 'hello']
my_list[1:]                     # ['hello', 3.14]
my_list[0:2]                    # [1, 'hello']
my_list[1:-1]                   # ['hello']
```

Lists

- Lists can be used to simulate a matrix

```
matrix = [
    [1, 2, 3],
    [4, 5, 6]
]

print(matrix[0][1])  # 2 (row 0, column 1)
print(matrix[1][2])  # 6 (row 1, column 2)

matrix[1][1] = 99
print(matrix)        # Output: [[1, 2, 3], [4, 99, 6]]
```

Lists

- Lists can be concatenated

```
my_list = [1, "hello", 3.14]

new_list = my_list + [5, "bye", True]
print(new_list)   # [1, "hello", 3.14, 5, "bye", True]
```

Lists

- Lists can be iterated with for or with special keyword enumerate

```
for i in [10, 20, 30, 40, 50]:
  print(i)                          # 10 20 30 40 50


nums= [10, 20, 30, 40, 50]
for index, value in enumerate(nums):
  print(index, value)
   # 0 10
   # 1 20
   # 2 30
   # 3 40
   # 4 50
```

Lists

- Enumerate can take a second parameter that set the index base, default is 0

```
nums= [10, 20, 30, 40, 50]
for index, value in enumerate(nums, 2):
  print(index, value)
   # 2 10
   # 3 20
   # 4 30
   # 5 40
   # 6 50
```

Lists

- To find out how many items are in a list the built-in len() function can be used

```
nums= [10, 20, 30, 40, 50]
print(len(nums))            # 5

empty_list = []
print(len(empty_list))      # 0

for i in range(len(nums)):
    print(num[i])
```

Lists

- Python lists aren’t just containers, they also come with built-in tools called methods that let you add, remove, or change items
- For adding one or multiple elements to a list are used append and extend methods

```
my_list = [1, "hello", 3.14]

my_list.append(10)              # [1, "hello", 3.14, 10]

my_list.extend([5, 6])          # [1, "hello", 3.14, 10, 5, 6]

my_list += ["new"]              # [1, "hello", 3.14, 10, 5, 6, "new"]
```

Lists

- To add an item at a specific position is used insert()

```
my_list = [1, "hello", 3.14]

my_list.insert(1, "add")           # [1, "add", "hello", 3.14]

my_list.insert(-1, 4)              # [1, "add", "hello", 4, 3.14]

my_list.insert(len(my_list), 12)  # [1, "add", "hello", 4, 3.14, 12]
```

Lists

- Adding or changing the value of an element can be done also directly

```
my_list = [1, "hello", 3.14]

my_list[1] = 21                    # [1, 21, 3.14]

my_list[2:] = ["A", "B", "C"]      # [1, 21, "A", "B", "C"]

my_list[:2] = [0]                  # [0, "A", "B", "C"]

my_list[1:3] = [10.2]              # [0, 10.2, "C"]
```

Lists

- To remove the first occurrence of an item (by value)

```
my_list = [1, "hello", 1, 3.14]

my_list.remove(1)              # ["hello", 1, 3.14]
my_list.remove(3.14)           # ["hello", 1]
```

Lists

- To remove an element from a specific position

```
my_list = [1, "hello", 1, 3.14, 5, 6, 7]

del my_list[2]          # [1, 'hello', 3.14, 5, 6, 7]
del my_list[-1]         # [1, 'hello', 3.14, 5, 6]
del my_list[:2]         # [3.14, 5, 6]
del my_list[0:2]        # [6]

del my_list
print(my_list)          # NameError
```

Lists

- Pop method removes and returns the item at the given index. If no index is given, it removes the last item

```
my_list = [1, "hello", 3.14]

my_list.pop()                   # [1, "hello"]
y = my_list.pop(1)              # y = "hello", my_list = [1]
y = my_list.pop(1000)           # IndexError
```

Lists

- To empty the entire list are two approaches:

```
my_list = [1, "hello", 1, 3.14, 5, 6, 7]
del my_list[:]        # []

my_list = [1, "hello", 1, 3.14, 5, 6, 7]
my_list.clear()       # []
```

Lists

- The assignment operator it’s not creating a new list, just a reference in memory to the same list
- The copy() method can be used to create shallow copy of a list (working for simple lists, not nested ones)

```
list1 = [1, 2, 3]
list2 = list1

list2.append(4)
print(list1)          # [1, 2, 3, 4]
print(list2)          # [1, 2, 3, 4]

list3 = list1.copy()
list3.append(5)
print(list3)          # [1, 2, 3, 4, 5]
```

Lists

- To find the index of the first occurrence of an item it’s used index method

```
my_list = [1, "hello", 3.14]

my_list.index("hello")  # 1
my_list.index(5)        # ValueError

check = 1 in my_list    # True
check = 5 in my_list    # False
```

Lists

- Count method it’s used to see how many times an item appears

```
my_list = [1, "hello", 3.14, 1]

x = my_list.count(1)    # 2
x = my_list.count(0)    # 0
```

Lists

- Reversing the order of a list

```
my_list = [1, "hello", 3.14]

my_list.reverse()              # [3.14, "hello", 1]
```

List Comprehension

List comprehension lets you create a new list by looping through an existing one, all in one line

```
[expression for item in iterable]
```

```
[x * 2 for x in [1, 2, 3]]  # [2, 4, 6]
```

List Comprehension

```
numbers = [0, 1, 2, 3, 4]
even_numbers = []

for number in numbers:
  if number % 2 == 0:
    even_numbers.append(number)
```

```
even_numbers = [number for number in range(5) if number % 2 == 0]
[0, 2, 4]
```

Lambda functions

What is a lambda function?

- A lambda function in Python is a small, anonymous function defined using the **lambda** keyword
- It can take any number of arguments but can only have a single expression
- The expression is evaluated and returned when the lambda function is called
- Lambda functions are often used for short, throwaway functions that are not needed elsewhere in the code

```
lambda <list of parameters> : return value
```

Lambda functions

- Without lambda

```
def add(a, b):
  return a+b

print(add(2, 3))
```

- With lambda

```
add = lambda a, b: a + b

print(add(2, 3))
```

Lambda functions

- more examples

```
repeat = lambda s: s * 3

print(repeat("Hi"))       # HiHiHi
print(repeat(4))          # 12
```

Lambda functions

- Lambda functions are created and bound at runtime. This means you can build a lambda with specific behavior while the program is running, based on data that's generated dynamically

```
def multiplier(n):
    return lambda x: x * n

three_times = multiplier(3)
seven_times = multiplier(7)
a = 5

print(three_times(a), seven_times(a))  # 15 35
```

Limitations

- Lambda functions are limited to a single expression, which can make them less readable for complex operations
- They do not have a name (unless assigned to a variable), which can make debugging more challenging

Built-in functions

Sort

- Sort method it’s used to sort elements from a list

```
sort(key=None, reverse=False)
```

```
cars = ['Ford', 'BMW', 'Volvo']
cars.sort()   # ['BMW', 'Ford', 'Volvo']
```

Sort

```
def myFunc(e):
  return e['year']


cars = [
    {'car': 'Ford', 'year': 2005},
    {'car': 'Mitsubishi', 'year': 2000},
    {'car': 'BMW', 'year': 2019},
    {'car': 'VW', 'year': 2011}
]
cars.sort(key=myFunc) # [{'car': 'Mitsubishi', 'year': 2000},
                      #  {'car': 'Ford', 'year': 2005},
                      #  {'car': 'VW', 'year': 2011},
                      #  {'car': 'BMW', 'year': 2019}]
```

Sorted

- Sorted method returns a new sorted list from the elements of any iterable. It does not change the original iterable

```
sorted(iterable, key=None, reverse=False)

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

y = sorted(x)                                    # [1, 2, 3, 4, 5]
y = sorted(x, reverse=True)                      # [5, 4, 3, 2, 1]
y = sorted(x, key = lambda i: i%3)               # [3, 1, 4, 2, 5]
y = sorted(x, key = lambda i: i%3, reverse=True) # [2, 5, 1, 4, 3]
```

Reversed

- Reversed returns a reversed iterator of the given sequence (like a list, string, tuple), without changing the original

```
reversed(sequence)

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

y = list(reversed(x))   # [5, 3, 4, 1, 2]
```

Filter

- Filter function it’s used to filter elements from an iterable based on a condition

```
filter(function, iterable)
```

function - a function that returns True or False for each element.
iterable - the list to be filtered.

The result will be a filter object, which can be converted into a list.

Filter

```
x = [1, 2, 3, 4, 5]
y = list(filter(lambda element: element % 2 == 0, x))  # [2, 4]


def myFunc(x):
  if x < 3:
    return False
  else:
    return True

y = list(filter(myFunc, x))  # [3, 4, 5]
```

Map

- Map applies a function to every item in an iterable and returns a new iterable with the transformed items

```
map(function, iterable)
```

function - a function that transform each element.
iterable - the list of elements to apply the function.

The result will be a map object, which can be converted into a list.

Map

```
x = [1, 2, 3]
y = map(lambda element: element*element, x)          # y will be an iterable object
list(y)                                              # [1, 4, 9]


def myfunc(n):
  return len(n)

x = list(map(myfunc, ['apple', 'banana', 'cherry'])) # [5, 6, 6]
```

Map/Filter together with range

- Both filter and map can also be used to create a list (usually in conjunction with range)

```
range(start, stop, step)
```

```
x = list(map(lambda x: x*x, range(1, 10)))
# [1, 4, 9, 16, 25, 36, 49, 64, 81]

x = list(filter(lambda x: x%7 == 1, range(1, 100)))
# [1, 8, 15, 22, 29, 36, 43, 50, 57, 64, 71, 78, 85, 92, 99]
```

Zip

- Zip it’s used to group two or more iterable objects into one iterable object. Zip with * character it’s used to unzip such a list

```
zip(iterable1, iterable2, …)
```

```
x = [1, 2, 3]
y = [10, 20, 30]

z = list(zip(x, y)) # [(1, 10), (2, 20), (3, 30)]

my_list = [(1, 2) , (3 ,4) , (5, 6)]
a, b = zip(*my_list)
print(a)  # (1, 3, 5)
print(b)  # (2, 4, 6)
```

Reduce

- Reduce function reduces a list (or other iterable) to a single value, by applying a function cumulatively to the elements

```
reduce(function, iterable, initializer)
```

```
from functools import reduce

def add(x, y):
    return x + y

x = [1, 2, 3, 4, 5]
y = reduce(add, x)   # 15

z = reduce(lambda x, y: x * y, [1, 2, 3], 10)  # 60
```

Max & Min

```
max(iterable, [key])
max(el1, el2, el3, …)
```

```
min(iterable, [key])
min(el1, el2, el3, …)
```

```
x = [1, 2, 3, 4, 5]
y = max(x)                          # 5
y = max(1, 3, 2, 7, 9, 3, 5)        # 9
y = max(x, key = lambda i: i % 3) # 2
```

```
x = [1, 2, 3, 4, 5]
y = min(x)                          # 1
y = min(1, 3, 2, 0, 9, -3, 5)       # -3
y = min(x, key = lambda i: i % 3) # 3
```

Sum

- adds up all the numerical values in an iterable

```
sum(iterable, [start])
```

```
x = [1, 2, 3, 4, 5]
y = sum(x)              # 15
y = sum(x,100)          # 115 (100+15)

x = [1, 2, "3", 4, 5]
y = sum(x)              # TypeError: unsupported operand type(s) for +:'int' and 'str'
```

Any & All

- built in functions for checking logical conditions

```
any([0, False, "", 3])                  # True – 3 it's also true
any([0, "", None, False])               # False – all are false

all([True, 1, "ok"])                    # True – all are true
all([True, 0, "text"])                  # False – 0 it's false

any(n % 2 == 0 for n in [1, 3, 5, 6])   # True – 6 is
all(n > 0 for n in [1, 5, 9, 0])        # False – 0 it's not grater than 0
```