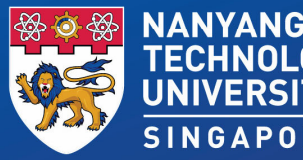

College of Computing and Data Science (CCDS)

CA6000 Applied AI Programming

Associate Professor
Nicholas Vun

aschvun@ntu.edu.sg

Built-in Data Structure in Python

Python Data Type Collections

For more complex requirement, four built-in data structures are provided in the Python programming language

- allow storing of multiple data and data types in a SINGLE variable

1. **List** - a collection which is ordered and changeable.
   Allows duplicate members.
2. **Tuple** - a collection which is ordered and unchangeable.
   Allows duplicate members.
3. **Dictionary** - a collection which is ordered and changeable.
   Does not allow duplicate members.
4. **Set** - a collection which is unordered, unchangeable, and unindexed.
   Does not allow duplicate members.

List

List is used when we need to store multiple strings or/and numbers.

It can be created by using square brackets [ ] separated by comma

```
fruits = ["apple", "banana", "cherry", "durian"]
print(fruits)
mixed = ["apple", 2, "cherry"]
print(mixed)
```

- useful for organizing and storing data sequences

List items are ordered
- items have a defined order, and that order will not change
- when new item is added to a list
  - the new items will be placed at the end of the list

But List items are changeable
- you can change, add, and remove items in a list after it has been created.

List manipulation

To access the item(s) in the list

- use index

```
fruits = ["apple", "banana", "cherry“, “durian”]
print(fruits[1])        #access item #1: banana
print(fruits[-1])       #access last item:durian
print(fruits[1:3])
```

Check for item in the list

```
fruits = ["apple", "banana", "cherry“, “durian”]
if “cherry" in fruits:
   print(“‘cherry' is in the fruits list")
```

List manipulation

To add item (to end of list)

```
fruits = ["apple", "banana", "cherry", "durian"]
fruits.append("orange")
print(fruits)
```

To insert item (without replacing item)

```
fruits.insert(1, "pineapple")
print(fruits)
```

To change item

```
fruits = ["apple", "banana", "cherry", "durian"]
fruits[1] = "orange"
print(fruits)
fruits[1:3] = ["pineapple","orange"]  #replace 2 items
print(fruits)
fruits[1:2] = ["mango","peach"]  #replace 1 item with 2 items
print(fruits)
```

To sort item

```
fruits.sort
print(fruits)
```

List manipulation

To extend the list with another list

```
fruits = ["apple", "banana", "cherry“]
local_fruits =[“pineapple”, “durian”]
fruits.extend(local_fruits)
print(fruits)
```

To remove item

```
fruits = ["apple", "banana", "cherry“, “durian”]
fruits.remove(“cherry”)
print(fruits)
```

To remove item based on specified index

```
fruits = ["apple", "banana", "cherry“, “durian”]
fruits.pop(2)
print(fruits)
```

List's Methods

append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Finger Exercise - List

Try this:

```
fruits = []    #create an empty list first

#ask for 3 types of fruit
for _ in range(3):    #single underscore instead of a variable
    fruits.append(input("Name a fruit: "))
print(fruits)

#sort the fruit alphabetically
for fruit in sorted(fruits):
    print(fruit)
```

`enumerate()` Function

The `enumerate()` function adds a counter to an iterable (such as List)

- returns it as an enumerate object (i.e. one by one)
- associate an index to each value (default starting at 0)

```
fruits = ["apple", "banana", "cherry"]

#enumerate the list
enu_fruits = enumerate(fruits, 10) #start with index = 10

#loop over an enumerate object to access the values of /
 each item
for index, item in emu_fruits:
    print(index)
    print(item)
```

Note that there is no need to manually increment the index in the loop

- i.e., enumerate enables ‘cleaner’ and easier to understand code

List Comprehension

List Comprehension is a piece of Python syntax that effectively lets you create list in a concise single line

- output the resulting elements to a newly constructed list

Example:

(a) To extract items to form a new list

```
fruits = ["apple", "banana", "cherry", "durian", "pineapple"]
new_list = []
for x in fruits:
    if "a" in x:
        new_list.append(x)
print(new_list)
```

(b) Using list comprehension

```
fruits = ["apple", "banana", "cherry", "durian", "pineapple"]
new_list = [x for x in fruits if "a" in x]
print(new_list)
```

List comprehension is very fast as it is written in c language

Finger Exercise - List Comprehension

Use List Comprehension to create a new list even_numbers from the list numbers :

```
numbers = [1, 12, 13, 24, 35, 38, 47]
even_numbers = [num                                    ]
print(even_numbers)
```

Tuple

A tuple is also a collection which is ordered like list but **unchangeable (immutable)**.

Tuples are written with round brackets ( ) separated by commas

- can have duplicate entries

```
fruits = ("apple", "banana", "cherry“, “banana”)
print(fruits)
```

Tuple can consist of different data types like list

```
tuple_list = ("abc", 34, True, 40, "male")
print(tuple_list[1]);
```

Tuple's methods

Does not have as many methods as compared to list

Examples:
Count the occurrence of a particular element

```
fruits = ("apple", "banana", "cherry“, “banana”)
banana = fruits.count(“banana”)
```

Find the index of the first occurrence of an element

```
first_banana = fruits.index(“banana”)
```

Tuple

A tuple is **immutable,** but …

- it can contain mutable elements such as list which you can change

```
# Tuple with mutable list
complex_tuple = ("apple", "banana", ["cherry“, “durian”])
```

- we can first change it to a list, made changes, change it back to tuple

```
this_tuple = ("apple", "banana", "cherry")
y = list(this_tuple) # change it to list
y.remove("apple")    # remove an entry
this_tuple = tuple(y)# change it back to tuple
```

Tuple - Summary

Tuple is fixed in size

- you can't change item in tuple
  - "safer" than List (defensive programming)
- execute faster than List

Some common use-cases include:

- Storing constants or configuration data
- Function return values with multiple components (see later)
- Dictionary keys (as they are hashable)

Dictionary

Index using number is used to access items in List and Tuple

- but we need to know beforehand where the items are stored
- not convenient

Dictionary allows us to dissociate item from index using number

- index using a key
- useful for storing data in a structured manner and accessing values by keys.

Dictionary stores data values in key:value pairs using the with curly braces { }

- does not allow duplicates

```
car  = {"brand": "BYD",
        "model": "SEAL",
        "year":  2024
       }
print(car)
print(len(car))
print(car["model"])
```

Dictionary

You can obtain the dictionary's keys

```
x = car.keys()
print(x)
```

Getting dictionary's values

```
print(car.values())
```

Getting dictionary's items (in the form of key:value pairs)

```
print(car.items())
```

Dictionary's items cannot have duplicate entries

- try the following

```
car = {  "brand": "BYD",
         "model": "SEAL",
         "year": 1964,
         "year": 2024
       }
print(car)
```

Dictionary

Dictionary is modifiable

```
car  = {"brand": "BYD",
        "model": "SEALION 7",
        "year": 2015,
       }

car["year"] = 2025      # change value
car["color"] = "RED"    # add new key and value

print(car.key)
print(car.values)


car.pop["model"]          # remove an entry
print(car.items())
```

Nested Dictionary

Nested ⇒ Dictionary within a Dictionary

```
cars = {“car_1”:{"brand": "Toyota",
                 "model": "Corolla",
                 "year": 2010
                },
        “car_2”:{"brand": “Mazda",
                 "model": "CX5",
                 "year": 2018
                },
        “car_3”:{"brand": "BYD",
                 "model": "SEAL",
                 "year": 2024
                }
       }

print(cars)
print(cars["car_1"]["model"])
```

Dictionary methods

Useful built-in methods for dictionary:

- `keys()` - Returns list of keys
- `values()` - Returns list of values
- `items()` - Returns (key, value) tuples
- `get()` - Returns value for key
- `pop()` - Removes key and returns value
- `update()` - Adds multiple key-values

Dictionary Comprehension

Similar to List Comprehension, Dictionary Comprehension allows us to create dictionaries in single line statement

Example: Given a fruits list, create a dictionary with fruit:len(fruit) key-value pairs.

(a)
```
fruits = ["apple", "banana", "cherry", "durian", "pineapple"]
fruit_lengths = {}  #an empty dictionary
for fruit in fruits:
    fruit_lengths[fruit] = len(fruit)
print(fruit_lengths)
```

(b) Using Dictionary comprehension
```
fruits = ["apple", "banana", "cherry", "durian", "pineapple"]
fruit_lengths = {fruit: len(fruit) for fruit in fruits}
print(fruit_lengths)
```

Set

Sets are written with curly brackets (i.e., like dictionary) but do not have the key-value pairs used by dictionary

- do not allow duplicate values
  - duplicate values will be ignored/remove automatically

```
this_set = {"apple","banana","cherry","banana"} #duplicate entries
this_set.add{"durian", "apple"} #add more entries
print(this_set)
```

Set items are unordered: i.e., the items in a set do not have a defined order.

- set items can appear in a different order every time you use them, and hence cannot be referred to by index or key.

```
#run the code twice to observe the change in display order
 this_set = {"apple","banana","cherry","banana","durian"}
 print(this_set)
```

Set

Set can be created from lists by using the `set()` constructors

```
mrts = [‘Jurong', ’Red Hill', ’Boon Lay', ’Jurong', ’Boon Lay’]
mrts_unique = list(set(Mrts))  #change back to list again
print(mrts_unique)
```

To create a new set containing all items from both sets: `union()` method can be used

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)
```

To insert one set to another set: `update()` method

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set1.unpdate(set2)
print(set1)
```

Set

Duplicate values will be ignored/remove automatically by set

- good for preventing/removing duplicate entry
  - without the need to write code to check for duplication

Set is also useful for testing membership and performing mathematical operations

a) `union()`
   return combined elements of two sets
b) `intersection()`
   find common elements of sets
c) `difference()`
   find the difference between sets

```
Set_1 = {1, 2, 3, 4}
Set_2 = {2, 3, 5, 6}
# Union
  print(Set_1 | Set_2 )
# Intersection
  print(Set_1 & Set_2 )
# Difference
  print(Set_1 - Set_2 )
  print(Set_1.difference(Set_2))
```

Summary – Python Data Structures

Lists [ ]- Ordered, mutable, allows duplicate elements

- Useful for storing sequences of data.

Tuples ( )- Ordered, immutable, allows duplicate elements.

- Equivalent to immutable lists
- Useful for storing sequence of data that are not to be changed

Dictionaries { } - Unordered, mutable, mapped by key-value pairs.

- Useful for storing data in a key-value format.

Sets { } - Unordered, mutable, contains unique elements

- Useful for membership testing and combining sets as it automatically eliminates duplicates