Computer Programming I

Binnur Kurt, PhD

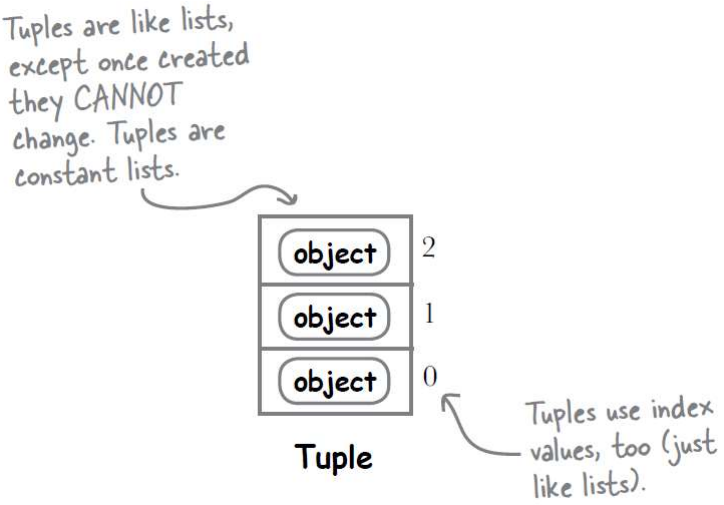
BAU Bahçeşehir University

binnur.kurt@rc.bau.edu.tr

MODULE 4

LISTS, TUPLES, SETS, AND DICTIONARIES

TUPLE



Tuple

- Tuples are immutable and typically store heterogeneous data, but the data can be homogeneous.
- A tuple's length is its number of elements and cannot change during program execution.

```
In [6]: names = ("jack", "kate", "james", "ben", "sun", "jin")
```

```
In [13]: names[0] = "jack shepherd"
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-f908c721d07b> in <module>
----> 1 names[0] = "jack shepherd"

TypeError: 'tuple' object does not support item assignment
```

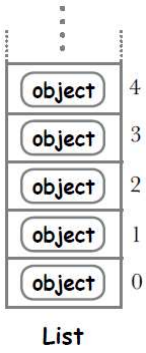
Tuple

```
In [6]: names = ("jack", "kate", "james", "ben", "sun", "jin")

In [7]: print(names)
        ('jack', 'kate', 'james', 'ben', 'sun', 'jin')

In [8]: print(names[0])
        ('jack', 'kate', 'james', 'ben', 'sun', 'jin')

In [10]: print(type(names))
         <class 'tuple'>

In [11]: print(len(names))
         6

In [12]: print(len(names[0]))
         4
```

Tuple

```
In [14]: sales = ("jack", "kate")
         finance = ( "james", "ben" )
         it = ("sun", "jin", "hugo")

In [19]: departments = (sales, finance, it)

In [20]: print(departments)
         (('jack', 'kate'), ('james', 'ben'), ('sun', 'jin', 'hugo'))

In [21]: print(len(departments))
         3

In [22]: print(departments[0])
         ('jack', 'kate')

In [23]: print(departments[1])
         ('james', 'ben')

In [24]: print(departments[2])
         ('sun', 'jin', 'hugo')
```

LIST



List

A list of the odd numbers

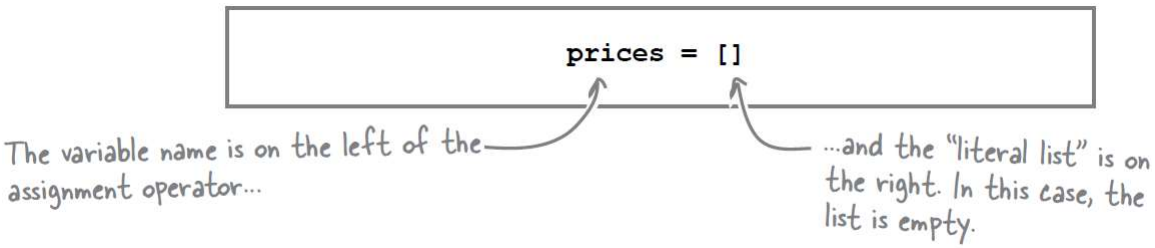
```
odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59]
```

The list starts with an opening square bracket.

The data values (a.k.a. "the objects") are separated from one another by a comma.

The list ends with a closing square bracket

Empty List



The variable name is on the left of the assignment operator...

...and the "literal list" is on the right. In this case, the list is empty.

Creating a List

```
temps = [ 32.0, 212.0, 0.0, 81.6, 100.0, 45.3 ]
```

Objects (in this case, some floats) are separated by commas and surrounded by square brackets—it's a list.

Creating a List

```
temps = [ 32.0, 212.0, 0.0, 81.6, 100.0, 45.3 ]
```

Objects (in this case, some floats) are separated by commas and surrounded by square brackets—it's a list.

```
words = [ 'hello', 'world' ]
```

A list of string objects

Creating a List

```
temps = [ 32.0, 212.0, 0.0, 81.6, 100.0, 45.3 ]
```

Objects (in this case, some floats) are separated by commas and surrounded by square brackets—it's a list.

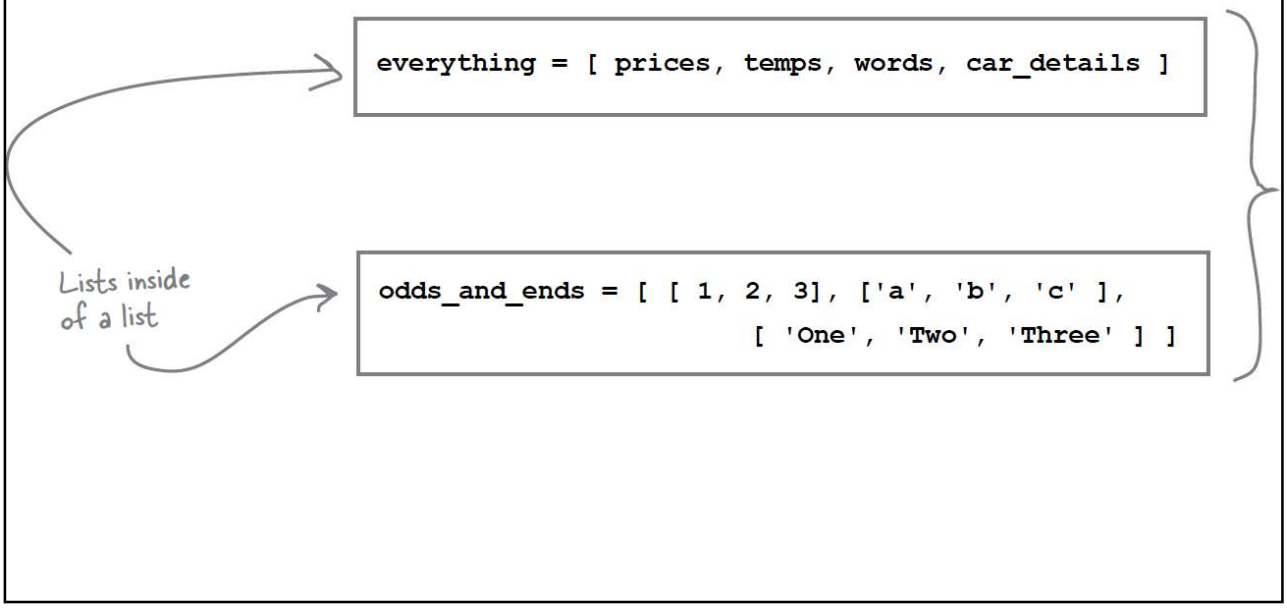
```
words = [ 'hello', 'world' ]
```

A list of string objects

```
car_details = [ 'Toyota', 'RAV4', 2.2, 60807 ]
```

A list of objects of differing type

Creating a List



List

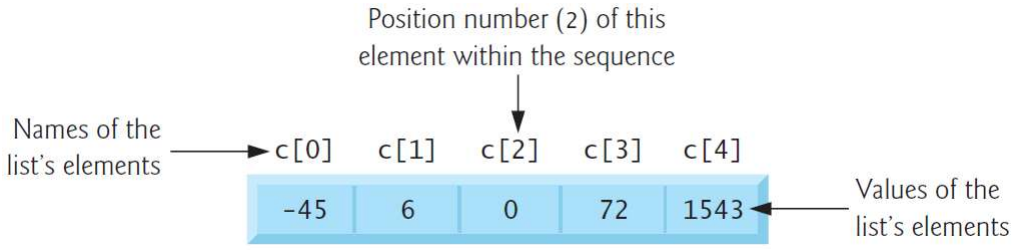
> Lists , like tuples, are sequences that contain elements referenced starting at zero.

> The individual elements of a list can be accessed in the same way as tuples.

```
In [133]: numbers
Out[133]: {0, 1, 2, 4, 5, 6, 7, 8, 9, 17}

In [134]: c = [-45, 6, 0, 72, 1543]
```

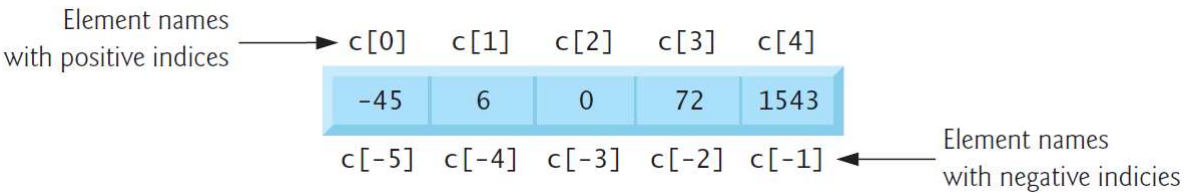
List

> You reference a list element by writing the list's name followed by the element's index enclosed in square brackets



Accessing Elements from the end with Negative Indices

> Lists also can be accessed from the end by using negative indices:



Lists Are Mutable

> Lists are mutable—their elements can be modified:

```
In [134]: c = [-45, 6, 0, 72, 1543]

In [135]: c[4] = 17

In [136]: c

Out[136]: [-45, 6, 0, 72, 17]
```

List: Examples (1/3)

```
In [25]: sales = ["jack", "kate"]
         finance = [ "james", "ben" ]
         it = ["sun", "jin", "hugo"]

In [26]: departments = [sales, finance, it]

In [27]: print(departments)
         [['jack', 'kate'], ['james', 'ben'], ['sun', 'jin', 'hugo']]

In [28]: print(len(departments))
         3

In [32]: departments.append(["sayid", "desmond", "charlie", "juliet"])

In [33]: print(departments)
         [['jack', 'kate'], ['james', 'ben'], ['sun', 'jin', 'hugo'], ['sayid', 'desmond', 'charlie', 'juliet']]

In [34]: print(len(departments))
         4
```

List: Examples (2/3)

```
In [29]: print(departments[0])
         ['jack', 'kate']

In [30]: print(departments[1])
         ['james', 'ben']

In [31]: print(departments[2])
         ['sun', 'jin', 'hugo']

In [35]: print(departments[3])
         ['sayid', 'desmond', 'charlie', 'juliet']
```

List: Examples (3/3)

```
In [37]: departments.extend([["miles", "claire", "libby"],["john", "walt", "eko", "ana lucia", "charlotte"]])

In [38]: print(len(departments))
         6

In [39]: print(departments)
         [['jack', 'kate'], ['james', 'ben'], ['sun', 'jin', 'hugo'], ['sayid', 'desmond', 'charlie', 'juliet'], ['miles', 'claire', 'li
         bby'], ['john', 'walt', 'eko', 'ana lucia', 'charlotte']]
```

Unpacking Sequences

> You can unpack any sequence's elements by assigning the sequence to a comma-separated list of variables

```
In [138]: jack= ("Jack", "Bauer", 1956, 75000 , True)

In [139]: first_name, last_name , birth_year, salary, fulltime = jack

In [146]: print(first_name, last_name, birth_year, salary, fulltime)
          Jack Bauer 1956 75000 True
```

Swapping Values Via Packing and Unpacking

> You can swap two variables' values using sequence packing and unpacking

```
In [148]: x = 3

In [147]: y = 5

In [149]: y , x = ( x , y )

In [151]: print(x , y)
          5 3
```

Accessing Indices and Values Safely with **`enumerate`**

- The preferred mechanism for accessing an element's index and value is the built-in function **`enumerate`**.
- This function receives an iterable and creates an iterator that, for each element, returns a tuple containing the element's index and value.

Accessing Indices and Values Safely with **`enumerate`**

```
In [152]: colors = ['red', 'orange', 'yellow']

In [153]: list(enumerate(colors))
Out[153]: [(0, 'red'), (1, 'orange'), (2, 'yellow')]

In [154]: tuple(enumerate(colors))
Out[154]: ((0, 'red'), (1, 'orange'), (2, 'yellow'))

In [155]: for index, color in enumerate(colors):
              print(f'{index}: {color}')
0: red
1: orange
2: yellow
```

Sequence Slicing

- You can slice sequences to create new sequences of the same type containing subsets of the original elements.
- Slice operations can modify mutable sequences—those that do not modify a sequence work identically for lists, tuples and strings

```
In [156]: numbers = [2, 3, 5, 7, 11, 13, 17, 19]

In [157]: numbers[2:6]
Out[157]: [5, 7, 11, 13]

In [158]: numbers[:6]
Out[158]: [2, 3, 5, 7, 11, 13]

In [159]: numbers[0:6]
Out[159]: [2, 3, 5, 7, 11, 13]

In [160]: numbers[6:]
Out[160]: [17, 19]

In [161]: numbers[6:len(numbers)]
Out[161]: [17, 19]

In [162]: numbers[:]
Out[162]: [2, 3, 5, 7, 11, 13, 17, 19]
```

Slicing with Steps

> The following code uses a step of 2 to create a slice with every other element of numbers

```
In [163]: numbers[::2]
Out[163]: [2, 5, 11, 17]
```

> You can use a negative step to select slices in reverse order.

```
In [164]: numbers[::-1]
Out[164]: [19, 17, 13, 11, 7, 5, 3, 2]

In [165]: numbers[-1:-9:-1]
Out[165]: [19, 17, 13, 11, 7, 5, 3, 2]
```

Modifying Lists Via Slices

> You can modify a list by assigning to a slice of it—the rest of the list is unchanged.

> The following code replaces numbers' first three elements, leaving the rest unchanged:

```
In [166]: numbers[0:3] = ['two', 'three', 'five']

In [168]: numbers
Out[168]: ['two', 'three', 'five', 7, 11, 13, 17, 19]
```

> The following deletes only the first three elements of numbers by assigning an empty list to the three-element slice:

```
In [169]: numbers[0:3] = []

In [170]: numbers
Out[170]: [7, 11, 13, 17, 19]
```

Modifying Lists Via Slices

> The following assigns a list's elements to a slice of every other element of numbers:

```
In [171]: numbers = [2, 3, 5, 7, 11, 13, 17, 19]

In [172]: numbers[::2] = [100, 100, 100, 100]

In [173]: numbers
Out[173]: [100, 3, 100, 7, 100, 13, 100, 19]

In [174]: id(numbers)
Out[174]: 1907384105344
```

Modifying Lists Via Slices

> Let's delete all the elements in numbers, leaving the existing list empty:

```
In [175]: numbers[:] = []

In [176]: numbers
Out[176]: []

In [177]: id(numbers)
Out[177]: 1907384105344
```

Modifying Lists Via Slices

> Deleting numbers' contents is different from assigning numbers a new empty list [] (snippet [22]).
> To prove this, we display numbers' identity after each operation.
> The identities are different, so they represent separate objects in memory:

```
In [178]: numbers = []

In [179]: numbers
Out[179]: []

In [180]: id(numbers)
Out[180]: 1907384105664
```

`del` Statement

> The del statement also can be used to remove elements from a list and to delete variables from the interactive session.
> You can remove the element at any valid index or the element(s) from any valid slice.
> Let's create a list, then use del to remove its last element:

```
In [181]: numbers = list(range(0, 10))

In [182]: numbers
Out[182]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [183]: del numbers[-1]

In [184]: numbers
Out[184]: [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

`del` Statement

> The following deletes the list's first two elements:

```
In [185]: del numbers[0:2]

In [186]: numbers
Out[186]: [2, 3, 4, 5, 6, 7, 8]
```

> The following uses a step in the slice to delete every other element from the entire list:

```
In [186]: numbers
Out[186]: [2, 3, 4, 5, 6, 7, 8]

In [187]: del numbers[::2]

In [188]: numbers
Out[188]: [3, 5, 7]
```

`del` Statement

> The following code deletes all the list's elements:

```
In [188]: numbers
Out[188]: [3, 5, 7]

In [189]: del numbers[:]

In [190]: numbers
Out[190]: []
```

Sorting a List in Ascending Order

> A common computing task called sorting enables you to arrange data either in ascending or descending order.

> List method sort modifies a list to arrange its elements in ascending order

```
In [191]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

In [192]: numbers.sort()

In [193]: numbers
Out[193]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Sorting a List in Descending Order

> To sort a list in descending order, call list method sort with the optional keyword argument **`reverse`** set to **`True`** (**`False`** is the default):

```
In [194]: numbers.sort(reverse=True)

In [196]: numbers
Out[196]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Built-In Function sorted

> Built-in function **`sorted`** ***returns a new list*** containing the sorted elements of its argument ***sequence***—the original sequence is ***unmodified***.

```
numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
```

```
ascending_numbers = sorted(numbers)
```

```
ascending_numbers
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
numbers
```

```
[10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
```

Key Functions

> Both list.sort() and sorted() have a key parameter to specify a function to be called on each list element prior to making comparisons.

```
names = ["jack", "Kate", "Ben", "james", "sun", "Jin"]
```

```
sorted(names)
```

```
['Ben', 'Jin', 'Kate', 'jack', 'james', 'sun']
```

```
sorted(names, key= str.lower)
```

```
['Ben', 'jack', 'james', 'Jin', 'Kate', 'sun']
```

Key Functions

```
stocks = [ ('orcl', 100.), ('gogle', 234.), ('ibm', 130.)]
```

```
sorted(stocks, key=lambda stock : stock[0])
```

```
[('gogle', 234.0), ('ibm', 130.0), ('orcl', 100.0)]
```

```
sorted(stocks, key=lambda stock : stock[1])
```

```
[('orcl', 100.0), ('ibm', 130.0), ('gogle', 234.0)]
```

```
sorted(stocks, key=lambda stock : stock[1], reverse= True)
```

```
[('gogle', 234.0), ('ibm', 130.0), ('orcl', 100.0)]
```

Searching Sequences

- Often, you'll want to determine whether a sequence (such as a list, tuple or string) contains a value that matches a key value.
- Searching is the process of locating a key.
- List method **`index`** takes as an argument a search key—the value to locate in the list—then searches through the list from index 0 and returns the index of the ***first*** element that matches the search key:

Searching Sequences

```
numbers = [3, 7, 1, 4, 2, 8, 5, 6]
```

```
numbers.index(5)
```

```
6
```

```
numbers.index(9)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-216-938fe671eb1a> in <module>
----> 1 numbers.index(9)

ValueError: 9 is not in list
```

Specifying the Starting Index of a Search

```
numbers
```

```
[3, 7, 1, 4, 2, 8, 5, 6]
```

```
numbers *= 2
```

```
numbers
```

```
[3, 7, 1, 4, 2, 8, 5, 6, 3, 7, 1, 4, 2, 8, 5, 6]
```

```
numbers.index(5,7)
```

```
14
```

Operators **`in`** and **`not in`**

> Operator in tests whether its right operand's iterable contains the left operand's value:

```
numbers
```

```
[3, 7, 1, 4, 2, 8, 5, 6, 3, 7, 1, 4, 2, 8, 5, 6]
```

```
42 in numbers
```

```
False
```

```
7 in numbers
```

```
True
```

```
42 not in numbers
```

```
True
```

```
7 not in numbers
```

```
False
```

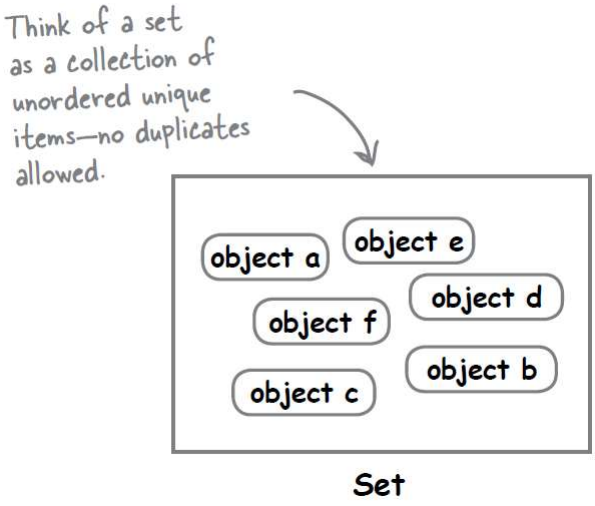
Using Operator **`in`** to prevent a **`ValueError`**

```
key = 42
```

```
if key in numbers:
    print(f'found {key} at index {numbers.index(key)}')
else:
    print(f'{key} is not found')
```

```
42 is not found
```

SET



Set

- A set is an unordered collection of unique values.
- Sets may contain only immutable objects, like strings, ints, floats and tuples that contain only immutable elements.
- Sets are iterable
- They are not sequences and do not support indexing and slicing with square brackets, [].
  - Dictionaries also do not support slicing.

Set

> The following code creates a set of strings named colors:

```
In [100]: colors = {'red', 'orange', 'yellow', 'green', 'red', 'blue'}

In [101]: len(colors)
Out[101]: 5
```

> Notice that the duplicate string 'red' was ignored without causing an error

Checking Whether a Value Is in a Set

> You can check whether a set contains a particular value using the in and not in operators:

```
In [102]: 'red' in colors
Out[102]: True

In [103]: 'purple' in colors
Out[103]: False

In [104]: 'purple' not in colors
Out[104]: True
```

Iterating Through a Set

> Sets are iterable, so you can process each set element with a for loop:

```
In [105]: for color in colors:
              print(color.upper(), end=' ')
YELLOW ORANGE BLUE GREEN RED
```

Comparing Sets

> The < operator tests whether the set to its left is a proper subset of the one to its right

```
In [106]: {1, 3, 5} == {3, 5, 1}
Out[106]: True

In [107]: {1, 3, 5} != {3, 5, 1}
Out[107]: False

In [108]: {1, 3, 5} < {3, 5, 1}
Out[108]: False

In [109]: {1, 3, 5} < {7, 3, 5, 1}
Out[109]: True
```

```
In [110]: {1, 3, 5}.issubset({3, 5, 1})
Out[110]: True

In [111]: {1, 2}.issubset({3, 5, 1})
Out[111]: False

In [112]: {1, 3, 5} > {3, 5, 1}
Out[112]: False

In [113]: {1, 3, 5, 7} > {3, 5, 1}
Out[113]: True
```

Mathematical Set Operations: Union

> The ***union*** of two sets is a set consisting of all the unique elements from both sets.

> You can calculate the union with the **|** operator or with the set type's **`union`** method:

```
In [114]: {1, 3, 5} | {2, 3, 4}
Out[114]: {1, 2, 3, 4, 5}

In [115]: {1, 3, 5}.union([20, 20, 3, 40, 40])
Out[115]: {1, 3, 5, 20, 40}
```

Mathematical Set Operations: Intersection

> The ***intersection*** of two sets is a set consisting of all the unique elements that the two sets have in common.

> You can calculate the intersection with the **&** operator or with the set type's **`intersection`** method:

```
In [116]: {1, 3, 5} & {2, 3, 4}
Out[116]: {3}

In [117]: {1, 3, 5}.intersection([1, 2, 2, 3, 3, 4, 4])
Out[117]: {1, 3}
```

Mathematical Set Operations: Difference

> The ***difference*** between two sets is a set consisting of the elements in the left operand that are not in the right operand.

> You can calculate the difference with the **-** operator or with the set type's **`difference`** method:

```
In [118]: {1, 3, 5} - {2, 3, 4}
Out[118]: {1, 5}

In [119]: {1, 3, 5, 7}.difference([2, 2, 3, 3, 4, 4])
Out[119]: {1, 5, 7}
```

Mathematical Set Operations: Symmetric Difference

> The ***symmetric difference*** between two sets is a set consisting of the elements of both sets that are not in common with one another.

> You can calculate the symmetric difference with the **^** operator or with the set type's **`symmetric_difference`** method:

```
In [120]: {1, 3, 5} ^ {2, 3, 4}
Out[120]: {1, 2, 4, 5}

In [121]: {1, 3, 5, 7}.symmetric_difference([2, 2, 3, 3, 4, 4])
Out[121]: {1, 2, 4, 5, 7}
```

Mathematical Set Operations: Disjoint

> Two sets are ***disjoint*** if they do not have any common elements.

> You can determine this with the set type's **`isdisjoint`** method:

```
In [122]: {1, 3, 5}.isdisjoint({2, 4, 6})
Out[122]: True

In [123]: {1, 3, 5}.isdisjoint({4, 6, 1})
Out[123]: False
```

Mutable Mathematical Set Operations

> Mutable Mathematical Set Operations like operator `|`, union augmented assignment `|=` performs a set union operation, but `|=` modifies its left operand:

```
In [124]: numbers = {1, 3, 5}

In [125]: numbers |= {2, 3, 4}

In [126]: numbers
Out[126]: {1, 2, 3, 4, 5}

In [127]: numbers.update(range(10))

In [128]: numbers
Out[128]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Mutable Mathematical Set Operations

- The other mutable set methods are:
  - intersection augmented assignment **&=**
  - difference augmented assignment **-=**
  - symmetric difference augmented assignment **^=**
- Their corresponding methods with iterable arguments are:
  - `intersection_update`
  - `difference_update`
  - `symmetric_difference_update`

Methods for Adding and Removing Elements from Set

- Set method **add** inserts its argument if the argument is not already in the set; otherwise, the set remains unchanged:

```
In [128]: numbers
Out[128]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

In [129]: numbers.add(17)

In [130]: numbers.add(3)

In [131]: numbers
Out[131]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 17}
```

Methods for Adding and Removing Elements from Set

> Set method **`remove`** removes its argument from the set—a KeyError occurs if the value is not in the set

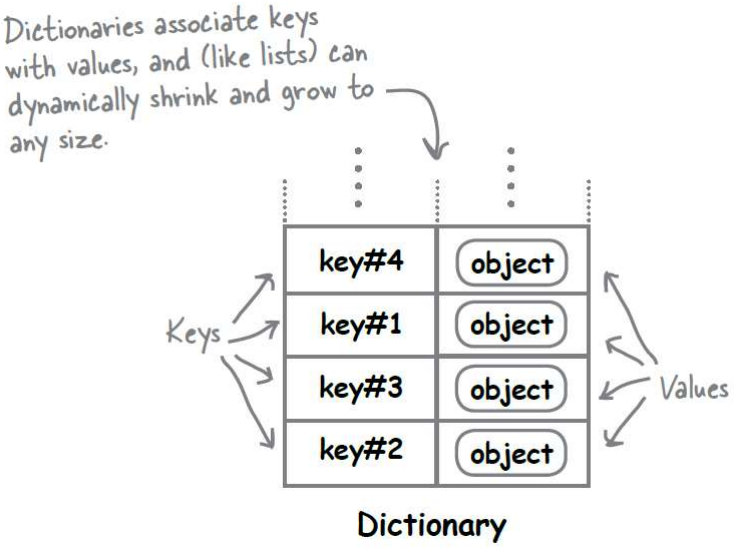
> Set method **`discard`** also removes its argument from the set but does not cause an exception if the value is not in the set.

```
In [131]: numbers
Out[131]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 17}

In [132]: numbers.remove(3)

In [133]: numbers
Out[133]: {0, 1, 2, 4, 5, 6, 7, 8, 9, 17}
```

Methods for Adding and Removing Elements from Set

> Set method **`discard`** also removes its argument from the set but does not cause an exception if the value is not in the set.

DICTIONARY



Dictionary

- A dictionary is like lists and tuples.
- It is another type of container for a group of data.
- Lists are indexed by their numeric order
- Dictionaries are indexed by names that you choose.
- These names can be letters, numbers, strings, or symbols — whatever suits you.

Unique Keys

> A dictionary's keys must be immutable (such as strings, numbers or tuples) and unique (that is, no duplicates).

> Multiple keys can have the same value, such as two different inventory codes that have the same quantity in stock.

Dictionary

```
In [41]: area_codes = {}
         area_codes["ankara"] = 312
         area_codes["istanbul"] = { "anadolu": 216, "avrupa": 212 }

In [42]: print(area_codes)

         {'ankara': 312, 'istanbul': {'anadolu': 216, 'avrupa': 212}}

In [43]: print(len(area_codes))

         2

In [44]: area_codes["istanbul"]["anadolu"]

Out[44]: 216

In [45]: area_codes["ankara"]

Out[45]: 312
```

Dictionary

```
In [46]: area_codes = { "ankara" : 312 , "istanbul" :  { "anadolu": 216, "avrupa": 212 } }

In [47]: print(area_codes)
         {'ankara': 312, 'istanbul': {'anadolu': 216, 'avrupa': 212}}
```

Dictionary

```
In [46]: area_codes = { "ankara" : 312 , "istanbul" :  { "anadolu": 216, "avrupa": 212 } }

In [47]: print(area_codes)
         {'ankara': 312, 'istanbul': {'anadolu': 216, 'avrupa': 212}}

In [48]: print(area_codes.values())
         dict_values([312, {'anadolu': 216, 'avrupa': 212}])

In [50]: print(area_codes.keys())
         dict_keys(['ankara', 'istanbul'])

In [51]: print(area_codes.items())
         dict_items([('ankara', 312), ('istanbul', {'anadolu': 216, 'avrupa': 212})])
```

Dictionary

```
In [65]: area_codes["antalya"] = 242

In [72]: area_codes.update({"bursa" : 224})

In [73]: print(area_codes)
         {'ankara': 312, 'istanbul': {'anadolu': 216, 'avrupa': 212}, 'antalya': 242, 'bursa': 224}

In [69]: area_codes.pop("bursa")
Out[69]: 224

In [70]: print(area_codes)
         {'ankara': 312, 'istanbul': {'anadolu': 216, 'avrupa': 212}, 'antalya': 242}
```

Dictionary

```
In [92]: for city in area_codes.keys() :
             if isinstance(area_codes[city],dict):
                for inner_city in area_codes[city].keys():
                     print("%s-%s -> %d " % (city, inner_city , area_codes[city][inner_city]))
             else:
                 print("%s -> %d " % (city, area_codes[city]))
         ankara -> 312
         istanbul-anadolu -> 216
         istanbul-avrupa -> 212
         antalya -> 242
         bursa -> 224
```

Converting Dictionary Keys, Values, and Pairs to Lists

```
In [76]: list(area_codes.keys())
Out[76]: ['ankara', 'istanbul', 'antalya', 'bursa']

In [77]: list(area_codes.values())
Out[77]: [312, {'anadolu': 216, 'avrupa': 212}, 242, 224]

In [78]: list(area_codes.items())
Out[78]: [('ankara', 312),
          ('istanbul', {'anadolu': 216, 'avrupa': 212}),
          ('antalya', 242),
          ('bursa', 224)]
```

Processing Keys in Sorted Order

```
In [80]: for city in sorted(area_codes.keys()) :
             print("%s -> %s" % (city,area_codes[city]))

ankara -> 312
antalya -> 242
bursa -> 224
istanbul -> {'anadolu': 216, 'avrupa': 212}
```

Iterating through a Dictionary

```
In [51]: print(area_codes.items())
```

```
dict_items([('ankara', 312), ('istanbul', {'anadolu': 216, 'avrupa': 212})])
```

```
In [54]: for city,code in area_codes.items() :
             print("%s -> %s " % (city,code))
```

```
ankara -> 312
istanbul -> {'anadolu': 216, 'avrupa': 212}
```

```
In [55]: for city in area_codes.keys() :
             print("%s -> %s " % (city,area_codes[city]))
```

```
ankara -> 312
istanbul -> {'anadolu': 216, 'avrupa': 212}
```

Standard Library Module **`collections`**

```
In [93]: from collections import Counter
```

```
In [96]: text = ('this is sample text with several words '
                 'this is more sample text with some different words')
```

```
In [97]: counter = Counter(text.split())
```

```
In [98]: for word, count in sorted(counter.items()):
             print(f'{word:<12}{count}')
```

```
different   1
is          2
more        1
sample      2
several     1
some        1
text        2
this        2
with        2
words       2
```