09. More data structures

October 1, 2025

1 9. More data structures

1.1 9.1. Dictionaries (type `dict`)

A dictionary is a data structure used to store collections of data in the form of key-value pairs. Each element in a dictionary is represented as a key-value pair. The key is used to uniquely identify the value associated with it. Keys are typically immutable data types like strings, numbers, or tuples (containing only immutable elements). Values can be of any data type, including lists, tuples, dictionaries, or even other objects.

1.1.1 9.1.1 Creating dictionaries

The empty dictionary Using a pair of curly braces:

[1]:
```python
my_dict = {}
type(my_dict)
```

[1]:
```
dict
```

Also, calling the `dict` constructor without parameters will create an empty dictionary:

[2]:
```python
my_other_dict = dict()
my_other_dict == my_dict
```

[2]:
```
True
```

Enumerating elements between curly braces Keys and values will be separated by a colon:

[3]:
```python
my_dict = {"name": "John", "age": 30, "city": "New York"}
```

1.1.2 9.1.2 Manipulating dictionaries

Getting an item from a dictionary Items in a dictionary can be retrieved by key:

[4]:
```python
my_dict["name"]
```

[4]:
```
'John'
```

Updating an item from a dictionary In order to update an item, assign a value for a certain key:

```
[5]: my_dict["name"] = "Mike"
     my_dict
```

```
[5]: {'name': 'Mike', 'age': 30, 'city': 'New York'}
```

Adding an item to a dictionary The same syntax we used for updating will add a new value to the dictionary, if the key doesn't exist:

```
[6]: my_dict["is_student"] = False
     my_dict
```

```
[6]: {'name': 'Mike', 'age': 30, 'city': 'New York', 'is_student': False}
```

Deleting an item from a dictionary Items can be deleted by key, using the keyword `del`:

```
[7]: del my_dict["is_student"]
     my_dict
```

```
[7]: {'name': 'Mike', 'age': 30, 'city': 'New York'}
```

Membership testing (`in` and `not in` operators) `in` and `not in` operators are used to check whether a key is in the dictionary or not:

```
[8]: "age" in my_dict
```

```
[8]: True
```

Getting the length of a dictionary Dictionaries are collections of items, so they have a length. Use `len` built-in to retrieve it.

```
[9]: len(my_dict)
```

```
[9]: 3
```

1.1.3 Exercises 9.1.1

1. Create a dictionary called `person` that contains the following key-value pairs: `python "name": "Jane" "age": 32 "occupation": "teacher" "hobbies": ["reading", "hiking"]`
   - print the value for key `age`
   - update the value for key `age` to 52
   - add a new item to the dictionary with key `"friends"` and value `["Anna", "Grace", "Mike"]`
   - add `"cooking"` to the hobbies list
   - print the dictionary and its length

1.1.4 9.1.2 Dictionary methods

d.get(key[, default]) Return the value for key if key is in the dictionary, else default. If default is not given, it defaults to None, so that this method never raises a KeyError.

```
[10]: my_dict.get("occupation", "N/A")
```

```
[10]: 'N/A'
```

d.pop(key[, default]) If key is in the dictionary, remove it and return its value, else return default. If default is not given and key is not in the dictionary, a KeyError is raised.

```
[11]: city = my_dict.pop("city")
```

```
[12]: city
```

```
[12]: 'New York'
```

```
[13]: my_dict
```

```
[13]: {'name': 'Mike', 'age': 30}
```

d.update(other) Update d using items in other, overwriting existing keys. update() accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two).

```
[14]: my_dict.update({"age": 20, "occupation": "student"})
```

```
[15]: my_dict
```

```
[15]: {'name': 'Mike', 'age': 20, 'occupation': 'student'}
```

d.items() Returns items in dictionary, as (key, value) tuples. Used for iterating on the dictionary:

```
[16]: for key, value in my_dict.items():
          print(key, value)
```

```
name Mike
age 20
occupation student
```

d.keys() Returns keys in dictionary; used for iterating on the dictionary, when only keys are needed:

```
[17]: for key in my_dict.keys():
          print(key)
```

```
name
age
occupation
```

`d.values()` Returns values in dictionary; used for iterating on the dictionary, when only values are needed:

[18]:
```python
for value in my_dict.values():
    print(value)
```

```
Mike
20
student
```

1.1.5 Exercises 9.1.2

1. Using the `person` dictionary above:
    1. Remove key `occupation` and save its value in a variable; print the variable.
    2. Try getting value key `height` (or `None`, if key does not exist) from the dictionary and store it in a variable; print the variable.
    3. Update the dictionary with the following dictionary: `measurements = {"weight": 75, "height": 1.78, "name": "John"}`
    4. Iterate on the dictionary to print its keys and values.
2. Given a list of strings build a dictionary that has each unique string as a key and the number of appearances as a value. E.g. `['hello', 'hello', 'is', 'there', 'anybody', 'in', 'there']` -> `{'hello': 2, 'is': 1, 'there': 2, 'anybody': 1, 'in': 1}`

1.2 9.2 Sets (type `set`)

Sets are unordered collections of unique elements.

They can be constructed using their type constructors: * Empty set: `set()`, `frozenset()` * From an iterable: `set(iterable)`, `frozenset(iterable)` * Using curly brackets, separating items with commas: `{a, b, c}`

[19]:
```python
s1 = set()
s2 = set(range(6))
s3 = set([1, 3, 5])
s4 = {4, 5, 6, 7, 8}
```

Instances of `set`:

`len(s)` Return the number of elements in set `s`.

[20]:
```python
len(s2)
```

[20]:
```
6
```

Membership tests

`x in s` Test x for membership in s.

`x not in s` Test x for non-membership in s.

```
[21]: 3 in s2
```

```
[21]: True
```

```
[22]: 0 not in s1
```

```
[22]: True
```

`s.isdisjoint(other)` Return `True` if the `s` has no elements in common with `other`. Sets are disjoint if and only if their intersection is the empty set.

```
[23]: s1.isdisjoint(s2)
```

```
[23]: True
```

`s.issubset(other)`

`s <= other` Test whether every element in `s` is in `other`.

`s < other` Test whether `s` is a proper subset of `other`, that is, `s <= other and s != other`.

```
[24]: s2 <= s3
```

```
[24]: False
```

```
[25]: s1 < s2
```

```
[25]: True
```

`s.issuperset(other)`

`s >= other` Test whether every element in `other` is in `s`.

`s > other` Test whether `s` is a proper superset of `other`, that is, `s >= other and s != other`.

```
[26]: s4.issuperset(s2)
```

```
[26]: False
```

```
[27]: s2 > s3
```

```
[27]: True
```

`s.union(other)`

s | other Return a new set with elements from s and other.

```
[28]: s2.union(s3)
```

```
[28]: {0, 1, 2, 3, 4, 5}
```

```
[29]: s3 | s4
```

```
[29]: {1, 3, 4, 5, 6, 7, 8}
```

s.intersection(other)

s & other Return a new set with elements common to s and other.

```
[30]: s2 & s3
```

```
[30]: {1, 3, 5}
```

s.difference(other)

s - other Return a new set with elements in s that are not other.

```
[31]: s2.difference(s3)
```

```
[31]: {0, 2, 4}
```

s.symmetric_difference(other)

s ^ other

```
[32]: s3 ^ s4
```

```
[32]: {1, 3, 4, 6, 7, 8}
```

There are also some operations available for set that do not apply to immutable instances of frozenset:

s.update(other)

s |= other Update s, adding elements from other.

```
[33]: s1.update(s2)
      s1
```

```
[33]: {0, 1, 2, 3, 4, 5}
```

s.intersection_update(other)

s &= other Update s, keeping only elements found in it and other.

[34]:
```
s1 &= s3
s1
```

[34]:
```
{1, 3, 5}
```

`s.difference_update(other)`

`s -= other` Update `s`, removing elements found in `other`.

[35]:
```
s1.difference_update(s4)
s1
```

[35]:
```
{1, 3}
```

`s.symmetric_difference_update(other)`

`s ^= other` Update `s`, keeping only elements found in either set, but not in both.

[36]:
```
s1 ^= s4
s1
```

[36]:
```
{1, 3, 4, 5, 6, 7, 8}
```

`s.add(elem)` Add element elem to the set.

[37]:
```
s1.add(9)
s1
```

[37]:
```
{1, 3, 4, 5, 6, 7, 8, 9}
```

`s.remove(elem)` Remove element `elem` from the set. Raises `KeyError` if `elem` is not contained in the set.

[38]:
```
s1.remove(8)
s1
```

[38]:
```
{1, 3, 4, 5, 6, 7, 9}
```

`s.discard(elem)` Remove element `elem` from the set if it is present.

[39]:
```
s1.discard(10)
s1
```

[39]:
```
{1, 3, 4, 5, 6, 7, 9}
```

`s.pop()` Remove and return an arbitrary element from the set. Raises `KeyError` if the set is empty.

[40]:
```
s1.pop()
```

[40]:
```
1
```

`s.clear()` Remove all elements from the set.

[41]:
```
s1.clear()
s1
```

[41]:
```
set()
```

1.3 Exercises 9.2

1. Create a set called `visited_cities` that contains the names of five cities you have visited in the past. Create a second set called `bucket_list` that contains the names of three cities you want absolutely want to visit.

   - Create the set `bucket_list_completed` which contains cities that are in both `visited_cities` and `bucket_list` (intersection).
   - Create the set `all_cities` which contains cities that are in either `visited_cities` or `bucket_list` (union).
   - Create the set `must_visit` which contains cities that are in `bucket_list`, but not in `visited_cities` (difference).

2. Write a Python program that counts the number of distinct words from a string. A word=a sequence of characters that is not whitespace (space, newline, tab).

   E.g. `python my_str = """beautiful is better than ugly  explicit is better than implicit  simple is better than complex  complex is better than complicated  flat is better than nested  sparse is better than dense"""  # Should print: 14 distinct words`