Other collections - Dictionaries and Sets

We’ve discussed three built-in sequence collections — `strings`, `lists` and `tuples`.

We’ve discussed three built-in sequence collections — `strings`, `lists` and `tuples`.

Now, we consider the built-in non-sequence collections — `dictionaries` and `sets`.

We've discussed three built-in sequence collections — `strings`, `lists` and `tuples`.

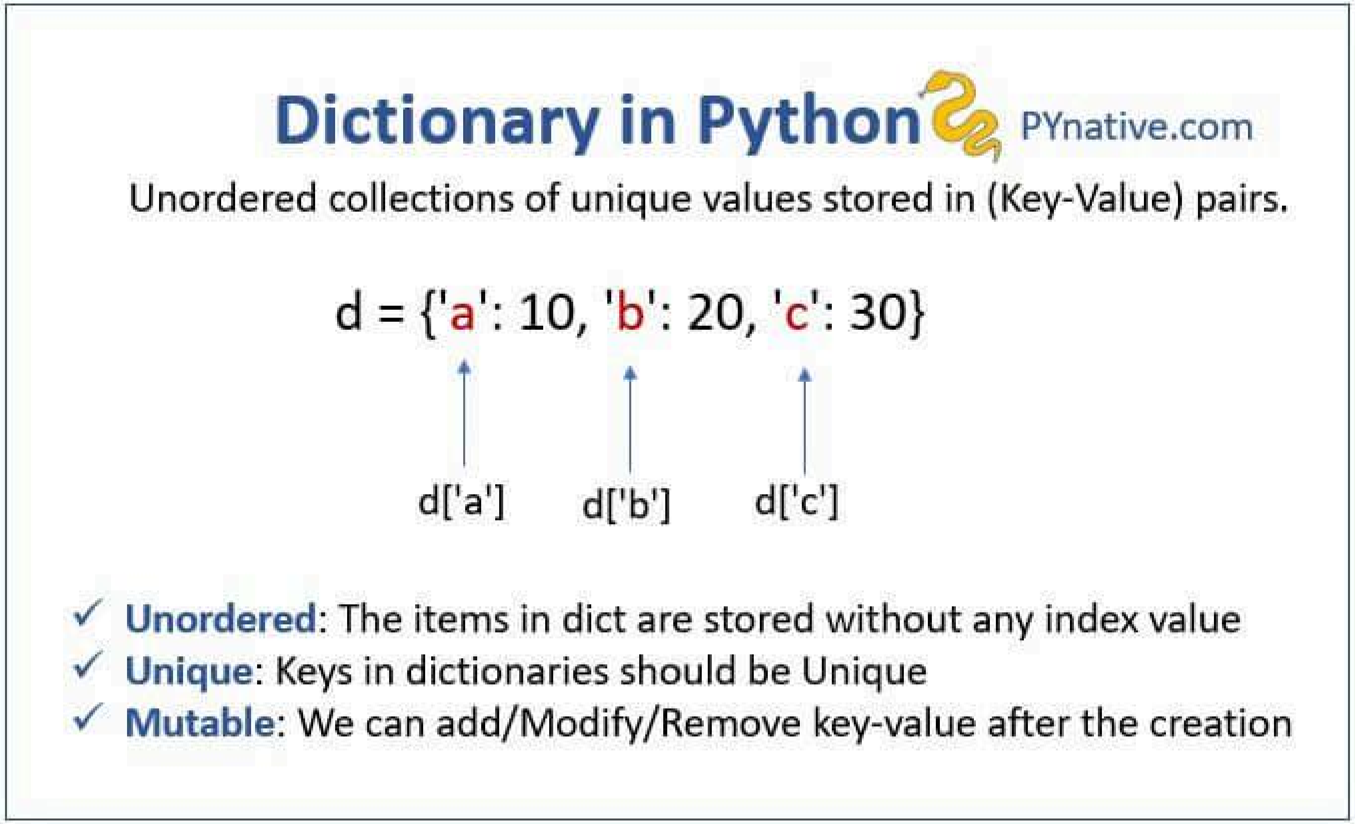
Now, we consider the built-in non-sequence collections — `dictionaries` and `sets`.

- A `dictionary` is an unordered collection which stores ***key–value*** pairs that map immutable keys to values, just as a conventional dictionary maps words to definitions.
- A `set` is an unordered collection of unique immutable elements.

Dictionaries

Like a `list`, a `dictionary` is a mutable collection of many values, but more general. In a `list`, the index positions have to be integers; in a `dictionary`, the indices can be any immutable data type.

Like a `list`, a `dictionary` is a mutable collection of many values, but more general. In a `list`, the index positions have to be integers; in a `dictionary`, the indices can be any immutable data type.


Dictionary in Python
PYnative.com
Unordered collections of unique values stored in (Key-Value) pairs.
d = {'a': 10, 'b': 20, 'c': 30}
d['a']
d['b']
d['c']
✓ Unordered: The items in dict are stored without any index value
✓ Unique: Keys in dictionaries should be Unique
✓ Mutable: We can add/Modify/Remove key-value after the creation

source: https://pynative.com/python-dictionaries/

You can think of a dictionary as a mapping between a set of indices (which are called ***keys***) and a set of ***values***. Each key maps to a value. The association of a key and a value is called a ***key-value*** pair or sometimes an ***item***.

You can think of a dictionary as a mapping between a set of indices (which are called ***keys***) and a set of ***values***. Each key maps to a value. The association of a key and a value is called a ***key-value*** pair or sometimes an ***item***.

A `dictionary`'s keys must be immutable (such as `strings`, `integers` or `tuples`) and unique (that is, no duplicates). However, multiple keys can have the same value.

You can think of a dictionary as a mapping between a set of indices (which are called ***keys***) and a set of ***values***. Each key maps to a value. The association of a key and a value is called a ***key-value*** pair or sometimes an ***item***.

A `dictionary`'s keys must be immutable (such as `strings`, `integers` or `tuples`) and unique (that is, no duplicates). However, multiple keys can have the same value.

As an example, we'll build a dictionary that maps from subjects to grades, so the keys are `string` while the values are `integers`. The function `dict` creates a new dictionary with no items.

```
In [2]: type({}) # {} also treated as dict in Python

Out[2]: dict
```

```
In [2]: type({}) # {} also treated as dict in Python

Out[2]: dict

In [3]: grade = dict()
        type(grade), grade

Out[3]: (dict, {})
```

```
In [2]: type({}) # {} also treated as dict in Python
```

```
Out[2]: dict
```

```
In [3]: grade = dict()
        type(grade), grade
```

```
Out[3]: (dict, {})
```

To add/update items to the dictionary, you can again use subscript operator (square brackets):

```
In [2]: type({}) # {} also treated as dict in Python

Out[2]: dict
```

```
In [3]: grade = dict()
        type(grade), grade

Out[3]: (dict, {})
```

To add/update items to the dictionary, you can again use subscript operator (square brackets):

```
In [4]: grade['calculus'] = 85 # Key:'calculus', value: 85
        print(grade) # Note that key and value are separate by colon

{'calculus': 85}
```

You can create a `dictionary` that contains multiple items by enclosing in curly braces, `{}`, a comma-separated list of key–value pairs, each of the form `key:value`.

You can create a `dictionary` that contains multiple items by enclosing in curly braces, `{}`, a comma-separated list of key–value pairs, each of the form `key:value`.

In [5]:
```
grade = {'calculus':85, 'introduction to mathematics':80, 'computer programmi
grade
```

Out[5]:
```
{'calculus': 85,
 'introduction to mathematics': 80,
 'computer programming': 90,
 'linear algebra': 95}
```

You can create a `dictionary` that contains multiple items by enclosing in curly braces, `{}`, a comma-separated list of key–value pairs, each of the form `key:value`.

In [5]:
```
grade = {'calculus':85, 'introduction to mathematics':80, 'computer programmi
grade
```

Out[5]:
```
{'calculus': 85,
 'introduction to mathematics': 80,
 'computer programming': 90,
 'linear algebra': 95}
```

Note that you can store them using separate `lists` for subjects and scores, but the following update and maintenance will become tedious:

```
subjects = ['calculus', 'introduction to mathematics', 'computer
programming', 'linear algebra']
score = [85, 80, 90, 95]
```

You can now use the keys to look up the corresponding values:

You can now use the keys to look up the corresponding values:

In [6]:
```
print(grade['computer programming'])
```

```
90
```

You can now use the keys to look up the corresponding values:

```
In [6]: print(grade['computer programming'])
```

```
90
```

You can obtain the number of items using `len()`

You can now use the keys to look up the corresponding values:

```
In [6]: print(grade['computer programming'])
```

```
90
```

You can obtain the number of items using `len()`

```
In [7]: len(grade)
```

```
Out[7]: 4
```

Trying to access a key that does not exist in a `dictionary` will result in a `KeyError` error message, much like a `list’s` “out-of-range” `IndexError` error message.

Trying to access a key that does not exist in a `dictionary` will result in a `KeyError` error message, much like a `list's` "out-of-range" `IndexError` error message.

```
In [8]: grade['English']
```

```
---------------------------------------------------------------------------
-----
KeyError                                  Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_35416\3219575328.py in <module>
----> 1 grade['English']

KeyError: 'English'
```

Trying to access a key that does not exist in a `dictionary` will result in a `KeyError` error message, much like a `list's` “out-of-range” `IndexError` error message.

```
In [8]: grade['English']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_35416\3219575328.py in <module>
----> 1 grade['English']

KeyError: 'English'
```

To add or delete an entry, it is similar to list

Trying to access a key that does not exist in a `dictionary` will result in a `KeyError` error message, much like a `list’s` “out-of-range” `IndexError` error message.

```
In [8]: grade['English']
```

```
---------------------------------------------------------------------------
-----
KeyError                                  Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_35416\3219575328.py in <module>
----> 1 grade['English']

KeyError: 'English'
```

To add or delete an entry, it is similar to list

```
In [9]: grade['English'] = 100
        grade
```

```
Out[9]: {'calculus': 85,
         'introduction to mathematics': 80,
         'computer programming': 90,
         'linear algebra': 95,
         'English': 100}
```

You can delete a key–value pair from a dictionary with the `del` statement:

You can delete a key–value pair from a dictionary with the `del` statement:

```
In [10]: del grade['English']
         grade
```

```
Out[10]: {'calculus': 85,
          'introduction to mathematics': 80,
          'computer programming': 90,
          'linear algebra': 95}
```

```
In [11]: display_quiz(path+"dict1.json", max_width=800)
```

What is printed by the following statements?

```
d1 = {"a": 1, "b": 2}
d2 = d1
d2["a"] = 99
print(d1)
```

{"a": 1, "b": 2}

{"a": 99, "b": 2}

An error occurs due to reassignment.

{"a": 99, "b": 2, "a": 1}

The `keys()`, `values()`, and `items()` Methods

There are three `dictionary` methods that will return `list`-like values of the `dictionary`'s keys, values, or both keys and values: `keys()`, `values()`, and `items()`.

There are three `dictionary` methods that will return `list`-like values of the `dictionary`'s keys, values, or both keys and values: `keys()`, `values()`, and `items()`.

The values returned by these methods are not true lists, but these data types (`dict_keys`, `dict_values`, and `dict_items`, respectively) can be used in `for` loops (Just like `range` object)!

There are three `dictionary` methods that will return `list`-like values of the `dictionary`'s keys, values, or both keys and values: `keys()`, `values()`, and `items()`.

The values returned by these methods are not true lists, but these data types (`dict_keys`, `dict_values`, and `dict_items`, respectively) can be used in `for` loops (Just like `range` object)!

If you want a true list from one of these methods, pass its list-like return value to the `list()` function

There are three `dictionary` methods that will return `list`-like values of the `dictionary`'s keys, values, or both keys and values: `keys()`, `values()`, and `items()`.

The values returned by these methods are not true lists, but these data types (`dict_keys`, `dict_values`, and `dict_items`, respectively) can be used in `for` loops (Just like `range` object)!

If you want a true list from one of these methods, pass its list-like return value to the `list()` function

In [12]:
```
subject = list(grade.keys())
score = list(grade.values())
print(subject)
print(score)
```

```
['calculus', 'introduction to mathematics', 'computer programming', 'l
inear algebra']
[85, 80, 90, 95]
```

```
In [13]: for v in grade.values():
             print(v)
```

```
85
80
90
95
```

```
In [13]: for v in grade.values():
             print(v)
```

```
85
80
90
95
```

Here, a `for` loop iterates over each of the values in the `grade` dictionary. A `for` loop can also iterate over the keys:

In [13]:
```
for v in grade.values():
    print(v)
```

```
85
80
90
95
```

Here, a `for` loop iterates over each of the values in the `grade` dictionary. A `for` loop can also iterate over the keys:

In [14]:
```
for k in grade.keys():
    print(k)
```

```
calculus
introduction to mathematics
computer programming
linear algebra
```

```
In [15]: for k in grade:
             print(k)
```

```
calculus
introduction to mathematics
computer programming
linear algebra
```

In [15]:
```python
for k in grade:
    print(k)
```

```
calculus
introduction to mathematics
computer programming
linear algebra
```

Note that by default, it will traverse over the keys!

In [15]:

```python
for k in grade:
    print(k)
```

```
calculus
introduction to mathematics
computer programming
linear algebra
```

Note that by default, it will traverse over the keys!

Dictionaries have a method called `items()` that returns a list of tuples, where each tuple is a key-value pair:

```
In [15]: for k in grade:
             print(k)
```

```
calculus
introduction to mathematics
computer programming
linear algebra
```

Note that by default, it will traverse over the keys!

Dictionaries have a method called `items()` that returns a list of tuples, where each tuple is a key-value pair:

```
In [16]: list(grade.items())
```

```
Out[16]: [('calculus', 85),
          ('introduction to mathematics', 80),
          ('computer programming', 90),
          ('linear algebra', 95)]
```

Combining `items()`, multiple assignment, and `for`, you can see a nice code pattern for traversing the keys and values of a dictionary in a single loop:

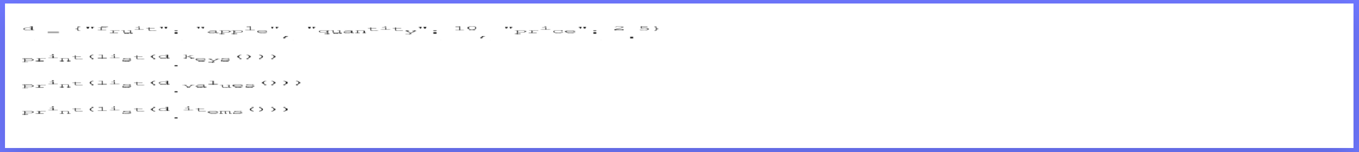
Combining `items()`, multiple assignment, and `for`, you can see a nice code pattern for traversing the keys and values of a dictionary in a single loop:

In [17]:
```
for key, val in grade.items():
    print(key,val)
```

```
calculus 85
introduction to mathematics 80
computer programming 90
linear algebra 95
```

In [18]:
```python
display_quiz(path+"dict2.json", max_width=800)
```

What is printed by the following statements?



```python
d = {"fruit": "apple", "quantity": 10, "price": 2.5}
print(list(d.keys()))
print(list(d.values()))
print(list(d.items()))
```

["fruit", "quantity", "price"] ["apple", 10, 2.5] ["fruit", "apple", "quantity", 10, "price", 2.5]

["fruit", "quantity", "price"] [("fruit", "apple"), ("quantity", 10), ("price", 2.5)] ["apple", 10, 2.5]

It prints the dictionary’s keys, values, and items without converting them to lists.

["fruit", "quantity", "price"] ["apple", 10, 2.5] [("fruit", "apple"), ("quantity", 10), ("price", 2.5)]

Checking Whether a Key or Value Exists in a Dictionary

Recall from the previous chapter that the `in` and `not in` operators can check whether a value exists in a list. You can also use these operators to see whether a certain key or value exists in a dictionary

Recall from the previous chapter that the `in` and `not in` operators can check whether a value exists in a list. You can also use these operators to see whether a certain key or value exists in a dictionary

```
In [19]: 'calculus' in grade, 'English' in grade.keys(), 85 in grade.values()
```

```
Out[19]: (True, False, True)
```

Retrieve value uisng `get()` Method

In [20]:
```
if 'English' in grade:
    e_score= grade['English']
```

In [20]:
```
if 'English' in grade:
    e_score= grade['English']
```

It’s tedious to check whether a key exists in a `dictionary` before accessing that key’s value. Fortunately, dictionaries have a `get()` method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.

```
In [20]: if 'English' in grade:
             e_score= grade['English']
```

It's tedious to check whether a key exists in a `dictionary` before accessing that key's value. Fortunately, dictionaries have a `get()` method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.

```
In [21]: picnicItems = {'apples': 5, 'cups': 2}
         print('I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.')
         print('I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.')
```

```
I am bringing 2 cups.
I am bringing 0 eggs.
```

Because there is no 'eggs' key in the `picnicItems` dictionary, the default value 0 is returned by the `get()` method. Without using `get()`, the code would have caused a `KeyError` message

Because there is no 'eggs' key in the `picnicItems` dictionary, the default value 0 is returned by the `get()` method. Without using `get()`, the code would have caused a `KeyError` message

In [22]:
```
picnicItems = {'apples': 5, 'cups': 2}
'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_35416\2486019301.py in <module>
      1 picnicItems = {'apples': 5, 'cups': 2}
----> 2 'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'

KeyError: 'eggs'
```

In [70]:
```python
display_quiz(path+"get.json", max_width=800)
```

What is printed by the following statements?

```python
d = {"name": "Alice", "age": 25}
print(d.get("name"))
print(d.get("gender", "Not Specified"))
```

Error

Alice Not Specified

None Not Specified

Alice None

Update value using `setdefault()` Method

You’ll often have to set a value in a `dictionary` for a certain key only if that key does not already have a value. The code looks something like this:

You'll often have to set a value in a `dictionary` for a certain key only if that key does not already have a value. The code looks something like this:

```
In [71]: spam = {'name': 'Pooka', 'age': 5}
         if 'color' not in spam:
             spam['color'] = 'black'
```

You'll often have to set a value in a `dictionary` for a certain key only if that key does not already have a value. The code looks something like this:

```
In [71]:
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

The `setdefault()` method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist.

The `setdefault()` method is a nice shortcut to ensure that a key exists. Here is a short program that counts the number of occurrences of each letter in a `string`.

The `setdefault()` method is a nice shortcut to ensure that a key exists. Here is a short program that counts the number of occurrences of each letter in a `string`.

In [23]:
```
message = 'It was a bright cold day in April, and the clocks were striking th
count = {}

for character in message:
    if character not in count:
        count[character] = 0
    count[character] = count[character] + 1

print(count)
```

```
{'I': 1, 't': 6, ' ': 13, 'w': 2, 'a': 4, 's': 3, 'b': 1, 'r': 5, 'i':
6, 'g': 2, 'h': 3, 'c': 3, 'o': 2, 'l': 3, 'd': 3, 'y': 1, 'n': 4,
'A': 1, 'p': 1, ',': 1, 'e': 5, 'k': 2, '.': 1}
```

In [24]:
```python
message = 'It was a bright cold day in April, and the clocks were striking th
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

print(count)
```

```
{'I': 1, 't': 6, ' ': 13, 'w': 2, 'a': 4, 's': 3, 'b': 1, 'r': 5, 'i':
6, 'g': 2, 'h': 3, 'c': 3, 'o': 2, 'l': 3, 'd': 3, 'y': 1, 'n': 4,
'A': 1, 'p': 1, ',': 1, 'e': 5, 'k': 2, '.': 1}
```

```
In [24]: message = 'It was a bright cold day in April, and the clocks were striking th
         count = {}

         for character in message:
             count.setdefault(character, 0)
             count[character] = count[character] + 1

         print(count)
```

```
{'I': 1, 't': 6, ' ': 13, 'w': 2, 'a': 4, 's': 3, 'b': 1, 'r': 5, 'i':
6, 'g': 2, 'h': 3, 'c': 3, 'o': 2, 'l': 3, 'd': 3, 'y': 1, 'n': 4,
'A': 1, 'p': 1, ',': 1, 'e': 5, 'k': 2, '.': 1}
```

You can view the execution of this program at https://autbor.com/setdefault. The program loops over each character in the `message` variable's string, counting how often each character appears. The `setdefault()` method ensures that the key is in the `count` `dictionary` (with a default value of 0) so the program doesn't throw a `KeyError` error when `count[character] = count[character] + 1` is executed!

In [24]:
```
message = 'It was a bright cold day in April, and the clocks were striking th
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

print(count)
```

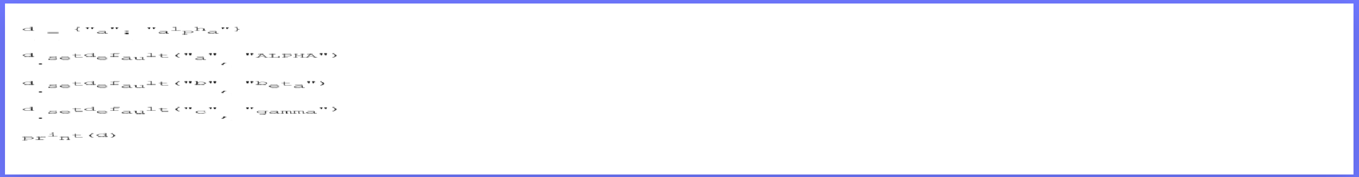
```
{'I': 1, 't': 6, ' ': 13, 'w': 2, 'a': 4, 's': 3, 'b': 1, 'r': 5, 'i':
6, 'g': 2, 'h': 3, 'c': 3, 'o': 2, 'l': 3, 'd': 3, 'y': 1, 'n': 4,
'A': 1, 'p': 1, ',': 1, 'e': 5, 'k': 2, '.': 1}
```

You can view the execution of this program at https://autbor.com/setdefault. The program loops over each character in the `message` variable's string, counting how often each character appears. The `setdefault()` method ensures that the key is in the `count dictionary` (with a default value of 0) so the program doesn't throw a `KeyError` error when `count[character] = count[character] + 1` is executed!

From the output, you can see that the lowercase letter c appears 3 times, the space character appears 13 times, and the uppercase letter A appears 1 time.

```
In [25]: display_quiz(path+"setdefault.json", max_width=800)
```

What is the final state of the dictionary 'd' after executing the following statements?



```
d = {"a": "alpha"}
d.setdefault("a", "ALPHA")
d.setdefault("b", "beta")
d.setdefault("c", "gamma")
print(d)
```

{"a": "alpha", "b": "beta"}	{"a": "ALPHA", "b": "beta", "c": "gamma"}
{"a": "alpha", "b": "beta", "c": "gamma"}	{"a": "ALPHA", "b": "beta"}

Dictionary Comprehensions

Dictionary comprehensions provide a convenient notation for quickly generating `dictionaries`, often by mapping one `dictionary` to another. For example, in a `dictionary` with unique values, you can reverse the key–value pairs:

Dictionary comprehensions provide a convenient notation for quickly generating `dictionaries`, often by mapping one `dictionary` to another. For example, in a `dictionary` with unique values, you can reverse the key–value pairs:

```
In [27]: months = {'January': 1, 'February': 2, 'March': 3}
```

Dictionary comprehensions provide a convenient notation for quickly generating `dictionaries`, often by mapping one `dictionary` to another. For example, in a `dictionary` with unique values, you can reverse the key–value pairs:

```
In [27]: months = {'January': 1, 'February': 2, 'March': 3}
```

```
In [28]: months2 = {number:name for name, number in months.items()}
         months2
```

```
Out[28]: {1: 'January', 2: 'February', 3: 'March'}
```

A dictionary comprehension also can map a `dictionary`'s values to new values. The following comprehension converts a `dictionary` of names and `lists` of grades into a `dictionary` of names and grade-point averages. The variables `k` and `v` commonly mean key and value:

A dictionary comprehension also can map a `dictionary`'s values to new values. The following comprehension converts a `dictionary` of names and `lists` of grades into a `dictionary` of names and grade-point averages. The variables `k` and `v` commonly mean key and value:

In [29]:
```
grades = {'Sue': [98, 87, 94], 'Bob': [84, 95, 91]}
```

A dictionary comprehension also can map a `dictionary`'s values to new values. The following comprehension converts a `dictionary` of names and `lists` of grades into a `dictionary` of names and grade-point averages. The variables `k` and `v` commonly mean key and value:

```
In [29]: grades = {'Sue': [98, 87, 94], 'Bob': [84, 95, 91]}
```

```
In [30]: grades2 = {k:sum(v)/len(v) for k, v in grades.items()}
         grades2
```

```
Out[30]: {'Sue': 93.0, 'Bob': 90.0}
```

Nested Dictionaries and Lists

A List of Dictionaries

Consider a game featuring aliens that can have different colors and point values. This simple `dictionary` stores information about a particular alien:

Consider a game featuring aliens that can have different colors and point values. This simple `dictionary` stores information about a particular alien:

```
In [36]: alien_0 = {'color': 'green', 'points': 5}
```

Consider a game featuring aliens that can have different colors and point values. This simple `dictionary` stores information about a particular alien:

```
In [36]: alien_0 = {'color': 'green', 'points': 5}
```

The `alien_0` dictionary contains a variety of information about one alien, but it has no room to store information about a second alien, much less a screen full of aliens. How can you manage a fleet of aliens? One way is to make a list of aliens in which each alien is a `dictionary` of information about that alien.

In [37]:

```python
aliens = []
# Make 30 green aliens.
for alien_number in range(30):
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)
# Show the first 5 aliens.
for alien in aliens[:5]:
    print(alien)
print("...")
# Show how many aliens have been created.
print(f"Total number of aliens: {len(aliens)}")
```

```
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
...
Total number of aliens: 30
```

In [37]:
```python
aliens = []
# Make 30 green aliens.
for alien_number in range(30):
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)
# Show the first 5 aliens.
for alien in aliens[:5]:
    print(alien)
print("...")
# Show how many aliens have been created.
print(f"Total number of aliens: {len(aliens)}")
```

```
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
...
Total number of aliens: 30
```

These aliens all have the same characteristics, but `Python` considers each one a separate object, which allows us to modify each alien individually. How might you work with a group of aliens like this?

Imagine that one aspect of a game has some aliens changing color and moving faster as the game progresses. When it’s time to change colors, we can use a `for` loop and an `if` statement to change the color of the aliens. For example, to change the first three aliens to yellow, medium-speed aliens worth 10 points each, we could do this:

Imagine that one aspect of a game has some aliens changing color and moving faster as the game progresses. When it's time to change colors, we can use a `for` loop and an `if` statement to change the color of the aliens. For example, to change the first three aliens to yellow, medium-speed aliens worth 10 points each, we could do this:

In [38]:
```
for alien in aliens[:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10

aliens[:10]
```

Out[38]:
```
[{'color': 'yellow', 'points': 10, 'speed': 'medium'},
 {'color': 'yellow', 'points': 10, 'speed': 'medium'},
 {'color': 'yellow', 'points': 10, 'speed': 'medium'},
 {'color': 'green', 'points': 5, 'speed': 'slow'},
 {'color': 'green', 'points': 5, 'speed': 'slow'},
 {'color': 'green', 'points': 5, 'speed': 'slow'},
 {'color': 'green', 'points': 5, 'speed': 'slow'},
 {'color': 'green', 'points': 5, 'speed': 'slow'},
 {'color': 'green', 'points': 5, 'speed': 'slow'},
 {'color': 'green', 'points': 5, 'speed': 'slow'}]
```

Sets

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.

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.

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

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.

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

```
In [39]: colors = {'red', 'orange', 'yellow', 'green', 'red', 'blue'} # Similar to set
         colors
```

```
Out[39]: {'blue', 'green', 'orange', 'red', 'yellow'}
```

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.

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

```
In [39]: colors = {'red', 'orange', 'yellow', 'green', 'red', 'blue'} # Similar to set
         colors
```

```
Out[39]: {'blue', 'green', 'orange', 'red', 'yellow'}
```

Notice that the duplicate string `'red'` was ignored (without causing an error). An important use of `sets` is **duplicate elimination**, which is automatic when creating a `set`. Also, the resulting `set’s` values may not be displayed in the same order as they were listed! Though the color names are displayed in sorted order, **sets are unordered**. You should not write code that depends on the order of their elements!

Though `sets` are iterable, they are not sequences and do not support indexing and slicing with square brackets, `[]`.

Though `sets` are iterable, they are not sequences and do not support indexing and slicing with square brackets, `[]`.

```
In [40]: colors[0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_35416\4110793329.py in <module>
----> 1 colors[0]

TypeError: 'set' object is not subscriptable
```

Though `sets` are iterable, they are not sequences and do not support indexing and slicing with square brackets, `[]`.

```
In [40]: colors[0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_35416\4110793329.py in <module>
----> 1 colors[0]

TypeError: 'set' object is not subscriptable
```

You can determine the number of items in a set with the built-in `len()` function:

Though `sets` are iterable, they are not sequences and do not support indexing and slicing with square brackets, `[]`.

```
In [40]: colors[0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_35416\4110793329.py in <module>
----> 1 colors[0]

TypeError: 'set' object is not subscriptable
```

You can determine the number of items in a set with the built-in `len()` function:

```
In [41]: len(colors)

Out[41]: 5
```

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

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

```
In [42]: 'red' in colors, 'purple' not in colors
```

```
Out[42]: (True, True)
```

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

```
In [42]: 'red' in colors, 'purple' not in colors
```

```
Out[42]: (True, True)
```

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

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

```
In [42]: 'red' in colors, 'purple' not in colors
```

```
Out[42]: (True, True)
```

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

```
In [43]: for color in colors: # {'blue', 'green', 'orange', 'red', 'yellow'}
             print(color, end=' ')
```

```
red yellow orange green blue
```

Creating a `Set` with the Built-In `set()` Function

You can create a `set` from another collection of values by using the built-in `set()` function — here we create a `list` that contains several duplicate integer values and use that `list` as `set`’s argument:

You can create a `set` from another collection of values by using the built-in `set()` function — here we create a `list` that contains several duplicate integer values and use that `list` as `set`'s argument:

```
In [44]: numbers = list(range(10)) + list(range(5))
         numbers
```

```
Out[44]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4]
```

You can create a `set` from another collection of values by using the built-in `set()` function — here we create a `list` that contains several duplicate integer values and use that `list` as `set`'s argument:

```
In [44]: numbers = list(range(10)) + list(range(5))
         numbers
```

```
Out[44]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4]
```

```
In [45]: set(numbers)
```

```
Out[45]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

If you need to create an empty `set`, you must use the `set()` function with empty parentheses, rather than empty braces, `{}`, which represent an empty `dictionary`:

If you need to create an empty `set`, you must use the `set()` function with empty parentheses, rather than empty braces, `{}`, which represent an empty `dictionary`:

```
In [46]: set()
```

```
Out[46]: set()
```

If you need to create an empty `set`, you must use the `set()` function with empty parentheses, rather than empty braces, `{}`, which represent an empty `dictionary`:

```
In [46]: set()
```

```
Out[46]: set()
```

Python displays an empty `set` as `set()` to avoid confusion with Python's string representation of an empty `dictionary` (`{}`).

`Set` Operators and Methods

`Sets` are mutable — you can add and remove elements, but set elements must be immutable. Therefore, a `set` cannot have other `sets` as elements.

`Sets` are mutable — you can add and remove elements, but set elements must be immutable. Therefore, a `set` cannot have other `sets` as elements.

```
In [47]: {6, 5, 'a'}
```

```
Out[47]: {5, 6, 'a'}
```

`Sets` are mutable — you can add and remove elements, but set elements must be immutable. Therefore, a `set` cannot have other `sets` as elements.

```
In [47]: {6, 5, 'a'}

Out[47]: {5, 6, 'a'}

In [48]: {7, 3, {3,5,7}}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_35416\3783331440.py in <module>
----> 1 {7, 3, {3,5,7}}

TypeError: unhashable type: 'set'
```

Methods for Adding and Removing Elements

Here we first discuss operators and methods that modify an existing `set`.

Here we first discuss operators and methods that modify an existing `set`.

Set method `update()` performs a union operation modifying the set in-place — the argument can be any iterable:

Here we first discuss operators and methods that modify an existing `set`.

Set method `update()` performs a union operation modifying the set in-place — the argument can be any iterable:

In [49]:
```
numbers = {1, 3, 5}
numbers.update(range(10))
numbers
```

Out[49]:
```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

`Set` method `add()` inserts its argument if the argument is not already in the set; otherwise, the `set` remains unchanged:

`Set` method `add()` inserts its argument if the argument is not already in the set; otherwise, the `set` remains unchanged:

```
In [50]: numbers.add(17)
         numbers.add(3)
         numbers
```

```
Out[50]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 17}
```

`Set` method `remove()` removes its argument from the `set` — a `KeyError` occurs if the value is not in the `set`:

`Set` method `remove()` removes its argument from the `set` — a `KeyError` occurs if the value is not in the `set`:

```
In [51]: numbers.remove(3)
         numbers
```

```
Out[51]: {0, 1, 2, 4, 5, 6, 7, 8, 9, 17}
```

`Set` method `remove()` removes its argument from the `set` — a `KeyError` occurs if the value is not in the `set`:

```
In [51]: numbers.remove(3)
         numbers
```

```
Out[51]: {0, 1, 2, 4, 5, 6, 7, 8, 9, 17}
```

```
In [52]: numbers.remove(11)
```

```
---------------------------------------------------------------------------
-----
KeyError                                  Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_35416\550142172.py in <module>
----> 1 numbers.remove(11)

KeyError: 11
```

In [53]:
```
display_quiz(path+"set.json", max_width=800)
```

Which of the following sets is equal to the set 's' after executing the following statements? (Remember that sets are unordered.)

```
s = {1, 2, 3, 4}
s.add(5)
s.update([3, 6, 7])
s.remove(4)
```

A KeyError is raised.

{1, 2, 3, 4, 5, 6, 7}

{1, 2, 3, 5, 6, 7}

{1, 2, 3, 4, 5}

Set Comprehensions

Like dictionary comprehensions, you define set comprehensions in curly braces. Let’s create a new `set` containing only the unique even values in the `list` numbers:

Like dictionary comprehensions, you define set comprehensions in curly braces. Let’s create a new `set` containing only the unique even values in the `list` numbers:

```
In [64]: numbers = [1, 2, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10, 10]
         evens = {item for item in numbers if item % 2 == 0}

         evens

Out[64]: {2, 4, 6, 8, 10}
```

Sorting the `set` and `dictionary`

As we mentioned last week, data types like `tuples` don’t provide methods like `sort()`. However `Python` provides the built-in function `sorted()`, which takes any sequence as a parameter and returns a new container with the same elements in a different order. You can also apply `sorted` to the set, but the returning container will be `list`.

As we mentioned last week, data types like `tuples` don't provide methods like `sort()`. However `Python` provides the built-in function `sorted()`, which takes any sequence as a parameter and returns a new container with the same elements in a different order. You can also apply `sorted` to the set, but the returning container will be `list`.

```
In [65]: help(sorted)
```

```
Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascend
ing order.

    A custom key function can be supplied to customize the sort order,
and the
    reverse flag can be set to request the result in descending order.
```

```
In [66]: RANKS = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]

         def rank_key(card):
             return RANKS.index(card)

         ori_set = {"A", "2", "7", "4", "Q"}
         print(sorted(ori_set))
         sorted_list = sorted(ori_set, key=rank_key)
         # Each element will be replaced by the output of rank_key() and sorts!
         print(sorted_list)
```

```
['2', '4', '7', 'A', 'Q']
['A', '2', '4', '7', 'Q']
```

In [66]:
```python
RANKS = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]

def rank_key(card):
    return RANKS.index(card)

ori_set = {"A", "2", "7", "4", "Q"}
print(sorted(ori_set))
sorted_list = sorted(ori_set, key=rank_key)
# Each element will be replaced by the output of rank_key() and sorts!
print(sorted_list)
```

```
['2', '4', '7', 'A', 'Q']
['A', '2', '4', '7', 'Q']
```

Note that we have changed the behavior of the `sorted()` function by providing the custom key that allows us to sort the data in a specific order using the predefined `list` and the `index()` function.

If you would like to sort the `dictionary`, you need to use the `items()` method (Otherwise, it will only return keys). The returning container will again be a `list`:

If you would like to sort the `dictionary`, you need to use the `items()` method (Otherwise, it will only return keys). The returning container will again be a `list`:

In [67]:
```python
grade = {'calculus':85, 'introduction to mathematics':80, 'introduction to co
sorted_list = sorted(grade.items())
print(sorted_list)
```

```
[('calculus', 85), ('introduction to computer science', 90), ('introdu
ction to mathematics', 80), ('linear algebra', 95)]
```

If you would like to sort the `dictionary`, you need to use the `items()` method (Otherwise, it will only return keys). The returning container will again be a `list`:

In [67]:
```
grade = {'calculus':85, 'introduction to mathematics':80, 'introduction to co
sorted_list = sorted(grade.items())
print(sorted_list)
```

```
[('calculus', 85), ('introduction to computer science', 90), ('introdu
ction to mathematics', 80), ('linear algebra', 95)]
```

If you would like to sort by the value, use the following code:

If you would like to sort the `dictionary`, you need to use the `items()` method (Otherwise, it will only return keys). The returning container will again be a `list`:

In [67]:
```
grade = {'calculus':85, 'introduction to mathematics':80, 'introduction to co
sorted_list = sorted(grade.items())
print(sorted_list)
```

```
[('calculus', 85), ('introduction to computer science', 90), ('introdu
ction to mathematics', 80), ('linear algebra', 95)]
```

If you would like to sort by the value, use the following code:

In [68]:
```
def value_key(x):
    return x[1]

grade = {'calculus':85, 'introduction to mathematics':80, 'computer programmi
sorted_dict = sorted(grade.items(), key=value_key)
print(sorted_dict)
```

```
[('introduction to mathematics', 80), ('calculus', 85), ('computer pro
gramming', 90), ('linear algebra', 95)]
```

> Exercise 2: Assume that we have a JSON file called `Pokemon.json`. Each item in the list is a dictionary that stores the name of the Pokemon as well as the species' strength. Complete the program by filling the missing part that reads the data of Pokemon from the `Pokemon.json` and displays the name with total species strength (summation of HP, Attack, Defense, Sp. Attack, Sp. Defense and Speed) line by line sorted by total species strength.

In [ ]:
```
import json

# 1. Load the JSON data from the file and reads contents to pokemon_data
with open('Pokemon.json', 'r') as file:
    pokemon_data = json.load(file)

# 2. Calculate the total species strength for each Pokemon and add it to poke
for pokemon in pokemon_data:
    # Your code here

    ______________________________
    pokemon['total_strength'] = total_strength

# 3. Sort the Pokemon data by total species strength
def value_key(x):
    return x['total_strength']

# Your code here
sorted_pokemon_data = sorted(pokemon_data, ________, ________)

# 4. Display the sorted Pokemon data
for pokemon in sorted_pokemon_data:
    print(________+ "has a total species strength of"+ ________)
```

In this chapter, we discussed Python’s `dictionary` and `set` collections. They are both unorder, mutable and do not allow duplicates.

A short comparison of the containers is shown below:

Feature	List	Tuple	Dictionary	Set
Mutable (Can be modified in place)	Yes	No	Yes (keys are immutable)	Yes
Iterable (Can be use in for loop)	Yes	Yes	Yes	Yes
Ordered (Can access by index, slicing)	Yes	Yes	No	No
Duplicate Values	Allowed	Allowed	Not in keys	Not allowed

```
In [69]: from jupytercards import display_flashcards
         fpath= "https://raw.githubusercontent.com/phonchi/nsysu-math106A/refs/heads/m
         display_flashcards(fpath + 'ch5.json')
```

key-value

Next

>