Python Dictionaries

- Dictionaries are used to store data values in key:value pairs
- Example
  - Dictionaries are written with curly brackets {}, and have keys and values

Dictionary Items

- Dictionary items are ordered, changeable, and does not allow duplicates.
- Dictionary items are presented in key:value pairs, and can be referred to by using the key name
- Example
  - Print the "brand" value of the dictionary

In [1]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
```

```
Ford
```

Ordered or Unordered?

- As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered
- When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.
- Unordered means that the items does not have a defined order, you cannot refer to an item by using an index

Changeable

- Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created

Duplicates Not Allowed

- Dictionaries cannot have two items with the same key

- Example
    - Duplicate values will overwrite existing values

In [2]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

Dictionary Length

- To determine how many items a dictionary has, use the len() function
- Example
    - Print the number of items in the dictionary

In [3]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(len(thisdict))
```

```
3
```

Dictionary Items - Data Types

- The values in dictionary items can be of any data type
- Example
    - String, int, boolean, and list data types

In [5]:
```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}

print(thisdict)
```

```
{'brand': 'Ford', 'electric': False, 'year': 1964, 'colors': ['red', 'wh
ite', 'blue']}
```

The dict() Constructor

- It is also possible to use the dict() constructor to make a dictionary

- Example
  - Using the dict() method to make a dictionary

In [4]:
```
thisdict = dict(name = "John", age = 36, country = "Norway")
print(thisdict)
```

```
{'name': 'John', 'age': 36, 'country': 'Norway'}
```

Accessing Items

- You can access the items of a dictionary by referring to its key name, inside square brackets [ ]
- Example
  - Get the value of the "model" key

In [7]:
```
thisdict =      {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
print(x)
```

```
Mustang
```

- There is also a method called get() that will give you the same result
- Example
  - Get the value of the "model" key

In [8]:
```
thisdict =      {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict.get("model")
print(x)
```

```
Mustang
```

Get Keys

- The keys() method will return a list of all the keys in the dictionary
- Example
  - Get a list of the keys

In [9]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

```
x = thisdict.keys()

print(x)
```

```
dict_keys(['brand', 'model', 'year'])
```

- The list of the keys is a view of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list
- Example
  - Add a new item to the original dictionary, and see that the keys list gets updated as well

In [10]:
```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.keys()

print(x) #before the change

car["color"] = "white"

print(x) #after the change
```

```
dict_keys(['brand', 'model', 'year'])
dict_keys(['brand', 'model', 'year', 'color'])
```

Get Values

- The values() method will return a list of all the values in the dictionary
- Example
  - Get a list of the values

In [11]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = thisdict.values()

print(x)
```

```
dict_values(['Ford', 'Mustang', 1964])
```

- The list of the values is a view of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list
- Example
  - Make a change in the original dictionary, and see that the values list gets updated as well

In [12]:
```python
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.values()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

```
dict_values(['Ford', 'Mustang', 1964])
dict_values(['Ford', 'Mustang', 2020])
```

- Example
  - Add a new item to the original dictionary, and see that the values list gets updated as well

In [13]:
```python
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.values()

print(x) #before the change

car["color"] = "red"

print(x) #after the change
```

```
dict_values(['Ford', 'Mustang', 1964])
dict_values(['Ford', 'Mustang', 1964, 'red'])
```

Get Items

- The items() method will return each item in a dictionary, as tuples in a list
- Example
  - Get a list of the key:value pairs

In [14]:
```python
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = thisdict.items()

print(x)
```

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
```

- The returned list is a view of the items of the dictionary, meaning that any changes done to the dictionary will be reflected in the items list
- Example
  - Make a change in the original dictionary, and see that the items list gets updated as well

In [15]:
```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.items()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 2020)])
```

- Example
  - Add a new item to the original dictionary, and see that the items list gets updated as well

In [16]:
```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.items()

print(x) #before the change

car["color"] = "red"

print(x) #after the change
```

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964), ('c
olor', 'red')])
```

Check if Key Exists

- To determine if a specified key is present in a dictionary use the in keyword
- Example
  - Check if "model" is present in the dictionary

In [17]:
```python
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

```
Yes, 'model' is one of the keys in the thisdict dictionary
```

Change Values

- You can change the value of a specific item by referring to its key name
- Example
    - Change the "year" to 2018

In [1]:
```python
thisdict =      {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

thisdict["year"] = 2018

print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2018}
```

Adding Items

- Adding an item to the dictionary is done by using a new index key and assigning a value to it
- Example

In [2]:
```python
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

Nested Dictionaries

- A dictionary can contain dictionaries, this is called nested dictionaries
- Example
    - Create a dictionary that contain three dictionaries

In [4]:
```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}

print(myfamily)
```

```
{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias',
'year': 2007}, 'child3': {'name': 'Linus', 'year': 2011}}
```

- Or, if you want to add three dictionaries into a new dictionary
- Example
    - Create three dictionaries, then create one dictionary that will contain the other three dictionaries

In [5]:
```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}

print(myfamily)
```

```
{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias',
'year': 2007}, 'child3': {'name': 'Linus', 'year': 2011}}
```

Access Items in Nested Dictionaries

- To access items from a nested dictionary, you use the name of the dictionaries, starting with the outer dictionary
- Example
    - Print the name of child 2

In [6]:
```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}

print(myfamily["child2"]["name"])
```

```
Tobias
```

Dictionary Methods

- Python has a set of built-in methods that you can use on dictionaries

- 1.clear()
- The clear() method removes all the elements from a dictionary
- Syntax - The dictionary.clear()
- No parameters
- Example
    - Remove all elements from the car list

In [7]:
```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

car.clear()

print(car)
```

```
{}
```

- 2.copy()
- The copy() method returns a copy of the specified dictionary
- Syntax - The dictionary.copy()
    - No parameters
- Example
    - Copy the car dictionary

In [8]:
```python
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.copy()

print(x)
```
```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

- 3.fromkeys()
- The fromkeys() method returns a dictionary with the specified keys and the specified value
- Syntax - The dict.fromkeys(keys, value)
  - keys - Required. An iterable specifying the keys of the new dictionary
  - value - Optional. The value for all keys. Default value is None
- Example
  - Create a dictionary with 3 keys, all with the value 0

In [9]:
```python
x = ('key1', 'key2', 'key3')
y = 0

thisdict = dict.fromkeys(x, y)

print(thisdict)
```
```
{'key1': 0, 'key2': 0, 'key3': 0}
```

- Example
  - Same example as above, but without specifying the value

In [10]:
```python
x = ('key1', 'key2', 'key3')

thisdict = dict.fromkeys(x)

print(thisdict)
```
```
{'key1': None, 'key2': None, 'key3': None}
```

- 4.pop()
- The pop() method removes the specified item from the dictionary
- The value of the removed item is the return value of the pop() method, see example below
- Syntax - The dictionary.pop(keyname, defaultvalue)
  - keyname - Required. The keyname of the item you want to remove
  - defaultvalue - Optional. A value to return if the specified key do not exist.

If this parameter is not specified, and the no item with the specified key is found, an error is raised

- Example
    - Remove "model" from the dictionary

In [11]:
```python
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

car.pop("model")

print(car)
```

```
{'brand': 'Ford', 'year': 1964}
```

- Example
    - The value of the removed item is the return value of the pop() method

In [12]:
```python
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.pop("model")

print(x)
```

```
Mustang
```

- 5.popitem()
- The popitem() method removes the item that was last inserted into the dictionary. In versions before 3.7, the popitem() method removes a random item
- The removed item is the return value of the popitem() method, as a tuple, see example below
- Syntax - The dictionary.popitem()
    - No parameters
- Example
    - Remove the last item from the dictionary

In [13]:
```python
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

car.popitem()

print(car)
```

```
{'brand': 'Ford', 'model': 'Mustang'}
```

- Example
    - The removed item is the return value of the popitem() method

In [14]:
```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.popitem()

print(x)
```

```
('year', 1964)
```

- 6.setdefault()
- The setdefault() method returns the value of the item with the specified key
- If the key does not exist, insert the key, with the specified value, see example below
- Syntax - The dictionary.setdefault(keyname, value)
    - keyname - Required. The keyname of the item you want to return the value from
    - value - Optional.If the key exist, this parameter has no effect.

If the key does not exist, this value becomes the key's value Default value None

- Example
    - Get the value of the "model" item

In [ ]:
```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.setdefault("model", "Bronco")

print(x)
```

- Example
    - Get the value of the "color" item, if the "color" item does not exist, insert "color" with the value "white"

In [16]:
```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.setdefault("color", "white")
```

```
print(car)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'white'}
```

- 7.update()
- The update() method inserts the specified items to the dictionary
- The specified items can be a dictionary, or an iterable object with key value pairs
- Syntax - The dictionary.update(iterable)
    - iterable A dictionary or an iterable object with key value pairs, that will be inserted to the dictionary
- Example
    - Insert an item to the dictionary

In [17]:
```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

car.update({"color": "White"})

print(car)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'White'}
```