Python Dictionaries

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

Dictionary

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

Dictionaries are written with curly brackets, and have keys and values:

In [3]:
```
# <!-- ExampleGet your own Python Server
# Create and print a dictionary:
#  -->
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

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

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.

In [4]:
```
# Example
# Print the "brand" value of the dictionary:

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:

In [5]:
```
# Example
# Duplicate values will overwrite existing values:

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:

In [6]:
```
# Example
# Print the number of items in the dictionary:

print(len(thisdict))
```

```
3
```

Dictionary Items - Data Types

The values in dictionary items can be of any data type:

In [7]:
```
# Example
# String, int, boolean, and list data types:

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

type()

From Python's perspective, dictionaries are defined as objects with the data type 'dict':

<class 'dict'>

In [8]:
```
# Example
# Print the data type of a dictionary:

thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(type(thisdict))
```

```
<class 'dict'>
```

The dict() Constructor

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

In [9]:
```
# Example
# Using the dict() method to make a dictionary:

thisdict = dict(name = "John", age = 36, country = "Norway")
print(thisdict)
```

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

Python Collections (Arrays)

There are four collection data types in the Python programming language:

- List is a collection which is ordered and changeable. Allows duplicate members.
- Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
- Set is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
- Dictionary is a collection which is ordered** and changeable. No duplicate members.
- Set items are unchangeable, but you can remove and/or add items whenever you like.

**As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

Python - Access Dictionary Items

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

In [10]:
```
# ExampleGet your own Python Server
# Get the value of the "model" key:

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

Out[10]:
```
'Mustang'
```

There is also a method called get() that will give you the same result:

In [11]:
```
# Example
# Get the value of the "model" key:

x = thisdict.get("model")
x
```

Out[11]:
```
'Mustang'
```

Get Keys

The keys() method will return a list of all the keys in the dictionary.

In [12]:
```
##### Example
# Get a list of the keys:

x = thisdict.keys()
x
```

Out[12]:
```
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.

In [13]:
```
# Example
# Add a new item to the original dictionary, and see that the keys
# list gets updated as well:

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

In [14]:
```
# Example
# Get a list of the values:

x = thisdict.values()
x
```

Out[14]:
```
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.

In [15]:
```
# Example
# Make a change in the original dictionary, and see that the values
# list gets updated as well:

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])
```

In [16]:
```
# Example
# Add a new item to the original dictionary, and see that the values
# list gets updated as well:

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.

In [17]:
```
# Example
# Get a list of the key:value pairs

x = thisdict.items()
# 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.
```

In [18]:
```
# Example
# Make a change in the original dictionary, and see that the items
# list gets updated as well:


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)])
```

In [19]:
```
# Example
# Add a new item to the original dictionary, and see that the items
# list gets updated as well:

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), ('colo
r', 'red')])
```

Check if Key Exists

To determine if a specified key is present in a dictionary use the in keyword:

In [20]:
```
# Example
# Check if "model" is present in the dictionary:

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
```

Python - Change Dictionary Items

Change Values

You can change the value of a specific item by referring to its key name:

In [21]:
```
# ExampleGet your own Python Server
# Change the "year" to 2018:

thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

thisdict["year"] = 2018
thisdict
```

Out[21]:
```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2018}
```

Update Dictionary

The update() method will update the dictionary with the items from the given argument.

The argument must be a dictionary, or an iterable object with key:value pairs.

In [22]:
```
# Example
# Update the "year" of the car by using the update() method:

thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
```

In [23]:
```
thisdict
```

Out[23]:
```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

Python - Add Dictionary Items

Adding Items

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

In [24]:
```
# Example
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

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

Update Dictionary

The update() method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.

The argument must be a dictionary, or an iterable object with key:value pairs.

In [35]:
```
# Example
# Add a color item to the dictionary by using the update() method:

thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"color": "red"})
```

In [26]:
```
thisdict
```

Out[26]:
```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

Python - Remove Dictionary Items

Removing Items

There are several methods to remove items from a dictionary:

In [27]:
```
# ExampleGet your own Python Server
# The pop() method removes the item with the specified key name:

thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

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

In [28]:
```
# Example
# The popitem() method removes the last inserted item
#(in versions before 3.7, a random item is removed instead):

thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

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

In [29]:
```
# Example
# The del keyword removes the item with the specified key name:

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

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

In [30]:
```
# Example
# The del keyword can also delete the dictionary completely:

thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict

print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_11332\1994730653.py in <module>
      9 del thisdict
     10
---> 11 print(thisdict) #this will cause an error because "thisdict" no longe
r exists.

NameError: name 'thisdict' is not defined
```

In [31]:
```
# Example
# The clear() method empties the dictionary:

thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

```
{}
```

Python - Loop Dictionaries

Loop Through a Dictionary

You can loop through a dictionary by using a for loop.

When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.

In [36]:
```
# Example
# Print all key names in the dictionary, one by one:

for x in thisdict:
  print(x)
```

```
brand
model
year
color
```

In [37]:
```
# Example
# Print all values in the dictionary, one by one:

for x in thisdict:
  print(thisdict[x])
```

```
Ford
Mustang
1964
red
```

In [38]:
```
# Example
# You can also use the values() method to return values of a dictionary:

for x in thisdict.values():
  print(x)
```

```
Ford
Mustang
1964
red
```

In [ ]:
```
# Example
# You can use the keys() method to return the keys of a dictionary:

for x in thisdict.keys():
  print(x)
```

In [39]:
```
# Example
# Loop through both keys and values, by using the items() method:

for x, y in thisdict.items():
  print(x, y)
```

```
brand Ford
model Mustang
year 1964
color red
```

Python - Copy Dictionaries

Copy a Dictionary

You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.

There are ways to make a copy, one way is to use the built-in Dictionary method copy().

In [40]:
```
# Example
# Make a copy of a dictionary with the copy() method:

thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

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

In [41]:
```
# Another way to make a copy is to use the built-in function dict().

# Example
# Make a copy of a dictionary with the dict() function:

thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

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

Python - Nested Dictionaries

Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries.

In [42]:
```
# Create a dictionary that contain three dictionaries:

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:

In [ ]:
```
# Example
# Create three dictionaries, then create one dictionary that will
# contain the other three dictionaries:

child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

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

Access Items in Nested Dictionaries

To access items from a nested dictionary, you use the name of the dictionaries, starting with the outer dictionary:

In [43]:
```
# Example
# Print the name of child 2:

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

```
Tobias
```

Python Dictionary Methods

Dictionary Methods

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

Method Description

- clear() Removes all the elements from the dictionary

- copy() Returns a copy of the dictionary

- fromkeys() Returns a dictionary with the specified keys and value

- get() Returns the value of the specified key

- items() Returns a list containing a tuple for each key value pair

- keys() Returns a list containing the dictionary's keys

- pop() Removes the element with the specified key

- popitem() Removes the last inserted key-value pair

- setdefault() Returns the value of the specified key. If the key does not ---

exist: insert the key, with the specified value

- update() Updates the dictionary with the specified key-value pairs

---

- values() Returns a list of all the values in the dictionary

In [ ]: