Dictionaries

Dictionaries are pretty much just like a dictionary in real life: they relate a value to another value. Formally, we say that there are *key-value pairs*: in other words, each *key* has an associated *value*. Each key must be unique and hashable (I'll talk more about this later), but there aren't really any restrictions on values. The use case for dictionaries is pretty much whenever you have a set of values that are associated with a unique set of keys.

Making a dictionary is quite simple:

Valid Keys

Technically speaking, keys can only be of a hashable data type. But what does it mean to be hashable? It means to have an hash value (id) that never changes no matter what. For simplicity's sake, you can just assume that you can only use immutable data types such as `int` and `str`. Lists and other dictionaries are not valid to be used as keys as they are mutable. If you are curious about the hash values, Example 8d-1 gives an example:

In [ ]:
```
# Example 8d-1: Hash values

mutable = [1, 2, 3]
print(id(mutable))

name = "Hi"
print(id(name))
```

```
2423909195456
2423907655216
```

`id()` is the built-in function to get a value's hash value. If you run this mutliple times without restarting the kernel, you will see that the list's id changes each time but the name's id never changes.

Making Dictionaries

Making a dictionary is quite simple:

In [ ]:
```
# Example 8d-2: Defining an empty dictionary

names = {}
```

Note that curly braces are used to make dictionaries. Do not use square braces, otherwise you would have made a list instead. If you wanted to instantiate a dictionary with some preset key-value pairs within it, you would insert pairs with the key and value separated by a colon and each pair separated by a comma:

In [ ]:
```
# Example 8d-3: Instantiating a dictionary with preset pairs

names = {"Anthony": "Pham", "Kaladin": "Stormblessed"}
```

Accessing Values

To access a value in a dictionary, the key must be known. The general syntax is `dict_name[key]`. This is shown in action in Example 8d-4:

In [ ]:
```
# Example 8d-4: Accessing a value in a key

names = {"Anthony": "Pham", "Kaladin": "Stormblessed"}

print(names["Anthony"])
print(names["Kaladin"])
```

```
Pham
Stormblessed
```

You must know the key beforehand in order to access the value. If you use an invalid key, you will get an `KeyError`. This is shown in Example 8d-5:

In [ ]:
```
# Example 8d-5: KeyError

names = {"Alana": "Shallan"}

print(names["Adolin"])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[27], line 5
      1 # Example 5: KeyError
      3 names = {"Alana": "Shallan"}
----> 5 print(names["Adolin"])

KeyError: 'Adolin'
```

If you just want all the values, then you can use `.values()` to obtain a `dict_values` of all the values. `dict_values` looks like a list, but it's not actually a list! If you're just looking to loop over all the keys, then you don't need to do anything else since `dict_values` can be iterated over. However, if you need an actual list of values then you can just cast to `list()`.

In [ ]:
```
# Example 8d-6: values() and iterable vs. list

couples = {"Adolin": "Shallan", "Dalinar": "Navani"}

values = couples.values()

print(values)
print(type(values))

for value in values:
    print(value, end=" ")


print()
list_values = list(values)
print(values)
print(type(list_values))
```

```
dict_values(['Shallan', 'Navani'])
<class 'dict_values'>
Shallan Navani
dict_values(['Shallan', 'Navani'])
<class 'list'>
```

Accessing Keys

Accessing keys requires you to use `.keys()` (if you want all the keys immediately) or to loop over the dictionary (if you want to go through each key one by one). This is shown in Example 8d-7:

In [ ]:
```
# Example 8d-7: Using keys() and a for loop

couples = {"Adolin": "Shallan", "Dalinar": "Navani"}

keys = couples.keys()

print(keys)
print(type(keys))

for key in couples:
    print(key, couples[key])
```

```
dict_keys(['Adolin', 'Dalinar'])
<class 'dict_keys'>
Adolin Shallan
Dalinar Navani
```

Note something important here: when you loop over a dictionary, it assumes you want the keys! This means you can simply loop over a dictionary to access each key-value pair one-by-one.

Adding to a dictionary

If you want to add to a dictionary, simply assign a value to a key that doesn't exist:

In [ ]:
```
# Example 8d-8: Adding to a dictonary

sauce_pairings = {"fries": "honey bbq", "salad": "ranch"}
print("Before addition: ", sauce_pairings)

sauce_pairings["milk"] = "hot sauce"
print("After addition: ", sauce_pairings)
```

```
Before addition:  {'fries': 'honey bbq', 'salad': 'ranch'}
After addition:  {'fries': 'honey bbq', 'salad': 'ranch', 'milk': 'hot sauce'}
```

I would like to clarify that I do not add hot sauce to my milk. I mostly abide by the food edition of the Geneva Conventions.

Editing a key-value pair

Editing a value assigned to a pre-exisitng key actually has the same format as adding to a dictionary, except the key already exists:

In [ ]:
```
# Example 8d-9: Editing a key-value pair

sauce_pairings = {"fries": "honey bbq", "salad": "ranch"}
print("Before edition: ", sauce_pairings)

sauce_pairings["fries"] = "hot sauce"
print("After edition: ", sauce_pairings)
```

```
Before edition:  {'fries': 'honey bbq', 'salad': 'ranch'}
After edition:  {'fries': 'hot sauce', 'salad': 'ranch'}
```

Note that you cannot directly edit a key. If you would like to assign a value to a new key, you would have to add a new pair (consisting of the desired value and the new key) and then remove the old key-value pair. Removing is shown in the next section.

Removing a key-value pair

There are two main ways to remove a key-value pair:

- `.pop()` - useful for when the removed value needs to be saved
- `del` - useful for when you don't care about the removed value

Both will do the same task, but the main difference is that `.pop()` will return the removed *value*. Note that `.pop()` is a function used on the dictionary while `del` is just a keyword. Example 8d-10 will demonstrate how to use `.pop()` and Example 8d-11 will demonstrate how to use `del`.

In [ ]:
```
# Example 8d-10: Using pop()

hobbies = {"anthony": "stuff", "alana": "reading"}
print("Before deletion: ", hobbies)

removed_value = hobbies.pop("alana")
print("After deletion: ", hobbies)
print("Removed value: ", removed_value)
```

```
Before deletion:  {'anthony': 'stuff', 'alana': 'reading'}
After deletion:  {'anthony': 'stuff'}
Removed value:  reading
```

In [ ]:
```
# Example 8d-11: Using del

hobbies = {"anthony": "stuff", "alana": "reading"}
print("Before deletion: ", hobbies)

del hobbies["alana"]
print("After deletion: ", hobbies)
```

```
Before deletion:  {'anthony': 'stuff', 'alana': 'reading'}
After deletion:  {'anthony': 'stuff'}
```

The argument for `.pop()` is just the key itself, and `del` must be followed up by `dict_name[key_to_be_removed]`. You can choose to not assign the value of `.pop()` to any value, but you cannot ever assign a `del` statement to a variable.