Python
Lecture 8

Dictionaries

Chapter 1

Introduction to Dictionary
A *dictionary* is an unordered set of `key: value` pairs.

It provides us with a way to map pieces of data to each other so that we can quickly find values that are associated with one another.

Suppose we want to store the prices of various items sold at a cafe:

- Avocado Toast is 6 dollars
- Carrot Juice is 5 dollars
- Blueberry Muffin is 2 dollars

In Python, we can create a dictionary called `menu` to store this data:

```
menu = {"avocado toast": 6, "carrot juice": 5, "blueberry muffin": 2
```

Notice that:

1. A dictionary begins and ends with curly braces `{` and `}`.
2. Each item consists of a key (`"avocado toast"`) and a value (`6`).
3. Each `key: value` pair is separated by a comma.

It's considered good practice to insert a space () after each comma, but our code will still run without the space.

Example

```
sensors =  {"living room": 21, "kitchen": 23, "bedroom": 20, "pantry": 22}
num_cameras = {"backyard": 6,  "garage": 2, "driveway": 1}

print(sensors)
```

Make a Dictionary
In the previous exercise, we saw a dictionary that maps strings to numbers (i.e., `"avocado toast": 6`). However, the keys can be numbers as well.

For example, if we were mapping restaurant bill subtotals to the bill total after tip, a dictionary could look like:

```
subtotal_to_total = {20: 24, 10: 12, 5: 6, 15: 18}
```

Values can be of any type. We can use a string, a number, a list, or even another dictionary as the value associated with a key!

For example:

```
students_in_classes = {"software design": ["Aaron", "Delila", 
"Samson"], "cartography": ["Christopher", "Juan", "Marco"], 
"philosophy": ["Frederica", "Manuel"]}
```

The list `["Aaron", "Delila", "Samson"]`, which is the value for the key `"software design"`, represents the students in that class.

We can also mix and match key and value types. For example:

```
person = {"name": "Shuri", "age": 18, "family": ["T'Chaka", 
"Ramonda"]}
```

Example

```
translations = {
  "mountain": "orod",
  "bread": "bass",
  "friend": "mellon",
  "horse": "roch"
}
print(translations)
```

Invalid Keys

We can have a list or a dictionary as a *value* of an item in a dictionary, but we cannot use these data types as keys of the dictionary. If we try to, we will get a `TypeError`.

For example:

```
powers = {[1, 2, 4, 8, 16]: 2, [1, 3, 9, 27, 81]: 3}
```

This code will yield:

```
TypeError: unhashable type: 'list'
```

The word “unhashable” in this context means that this ‘list’ is an object that can be changed.

Dictionaries in Python rely on each key having a *hash value*, a specific identifier for the key. If the key can change, that hash value would not be reliable. So the keys must always be unchangeable, hashable data types, like numbers or strings.

Empty Dictionary

A dictionary doesn’t have to contain anything. Sometimes we need to create an empty dictionary when we plan to fill it later based on some other input.

We can create an empty dictionary like this:

```
empty_dict = {}
```

We will explore ways to fill a dictionary in the next exercise.

Add A Key

To add a single `key: value` pair to a dictionary, we can use the syntax:

```
dictionary[key] = value
```

For example, if we had our `menu` dictionary from the first exercise:

```
menu = {"oatmeal": 3, "avocado toast": 6, "carrot juice": 5,
"blueberry muffin": 2}
```

And we wanted to add a new item, `"cheesecake"` for `8` dollars, we could use:

```
menu["cheesecake"] = 8
```

Now, `menu` looks like:

```
{"oatmeal": 3, "avocado toast": 6, "carrot juice": 5, "blueberry
muffin": 2, "cheesecake": 8}
```

Example

```
animals_in_zoo = {}
animals_in_zoo["zebras"] = 8
animals_in_zoo["monkeys"] = 12
animals_in_zoo["dinosaurs"] = 0
print(animals_in_zoo)
```

Add Multiple Keys

If we wanted to add multiple key : value pairs to a dictionary at once, we can use the `.update()` method.

Looking at our `sensors` object from a previous exercise:

```
sensors = {"living room": 21, "kitchen": 23, "bedroom": 20}
```

If we wanted to add 3 new rooms, we could use:

```
sensors.update({"pantry": 22, "guest room": 25, "patio": 34})
```

This would add all three items to the `sensors` dictionary.

Now, `sensors` looks like:

```
{"living room": 21, "kitchen": 23, "bedroom": 20, "pantry": 22,
"guest room": 25, "patio": 34}
```

Example

```
user_ids = {"teraCoder": 9018293, "proProgrammer": 119238}
user_ids.update({"theLooper": 138475, "stringQueen": 85739})
print(user_ids)
```

Overwrite Values
We know that we can add a key by using syntax like:

```
menu["avocado toast"] = 7
```

This will create a key `"avocado toast"` and set the value to `7`. But what if we already have an `'avocado toast'` entry in the `menu` dictionary?

In that case, our value assignment would overwrite the existing value attached to the key `'avocado toast'`.

```
menu = {"oatmeal": 3, "avocado toast": 6, "carrot juice": 5,
"blueberry muffin": 2}
menu["oatmeal"] = 5
print(menu)
```

This would yield:

```
{"oatmeal": 5, "avocado toast": 6, "carrot juice": 5, "blueberry
muffin": 2}
```

Notice the value of `"oatmeal"` has now changed to `5`.

Example

```
oscar_winners = {"Best Picture": "La La Land", "Best Actor": "Casey Affleck",
"Best Actress": "Emma Stone", "Animated Feature": "Zootopia"}

oscar_winners["Supporting Actress"] = "Viola Davis"
oscar_winners["Best Picture"] = "Moonlight"
```

Dict Comprehensions
Let’s say we have two lists that we want to combine into a dictionary, like a list of students and a list of their heights, in inches:

```
names = ['Jenny', 'Alexus', 'Sam', 'Grace']
heights = [61, 70, 67, 64]
```
Python allows you to create a dictionary using a dict comprehension, with this syntax:

```
students = {key:value for key, value in zip(names, heights)}
#students is now {'Jenny': 61, 'Alexus': 70, 'Sam': 67, 'Grace': 64}
```
Remember that `zip()` combines two lists into an iterator of tuples with the list elements paired together. This dict comprehension:

1. Takes a pair from the iterator of tuples
2. Names the elements in the pair `key` (the one originally from the `names` list) and `value` (the one originally from the `heights` list)
3. Creates a `key` : `value` item in the `students` dictionary
4. Repeats steps 1-3 for the entire iterator of pairs

Example

```
drinks = ["espresso", "chai", "decaf", "drip"]
caffeine = [64, 40, 0, 120]
zipped_drinks = zip(drinks, caffeine)
drinks_to_caffeine = {key:value for key, value in zipped_drinks}
print(drinks_to_caffeine)
```

Chapter 2

Using Dictionaries

Now that we know how to create a dictionary, we can start using already created dictionaries to solve problems.

In this lesson, you'll learn how to:

- Use a key to get a value from a dictionary
- Check for existence of keys
- Iterate through keys and values in dictionaries

Get A Key

Once you have a dictionary, you can access the values in it by providing the key. For example, let's imagine we have a dictionary that maps buildings to their heights, in meters:

```
building_heights = {"Burj Khalifa": 828, "Shanghai Tower": 632,
"Abraj Al Bait": 601, "Ping An": 599, "Lotte World Tower": 554.5,
"One World Trade": 541.3}
```

Then we can access the data in it like this:

```
>>> building_heights["Burj Khalifa"]
828
>>> building_heights["Ping An"]
599
```

Example

```
zodiac_elements = {"water": ["Cancer", "Scorpio", "Pisces"], "fire": ["Aries",
 "Leo", "Sagittarius"], "earth": ["Taurus", "Virgo", "Capricorn"], "air":["Gem
ini", "Libra", "Aquarius"]}

zodiac_elements["energy"] = "Not a Zodiac element"
print(zodiac_elements)
```

Get an Invalid Key

Let's say we have our dictionary of building heights from the last exercise:

```
building_heights = {"Burj Khalifa": 828, "Shanghai Tower": 632,
"Abraj Al Bait": 601, "Ping An": 599, "Lotte World Tower": 554.5,
"One World Trade": 541.3}
```

What if we wanted to know the height of the Landmark 81 in Ho Chi Minh City? We could try:

```
print(building_heights["Landmark 81"])
```

But `"Landmark 81"` does not exist as a key in the `building_heights` dictionary! So this will throw a KeyError:

```
KeyError: 'Landmark 81'
```

One way to avoid this error is to first check if the key exists in the dictionary:

```
key_to_check = "Landmark 81"

if key_to_check in building_heights:
  print(building_heights["Landmark 81"])
```

This will not throw an error, because `key_to_check in building_heights` will return `False`, and so we never try to access the key.

Try/Except to Get a Key

We saw that we can avoid `KeyError`s by checking if a key is in a dictionary first. Another method we could use is a `try/except`:

```
key_to_check = "Landmark 81"
try:
  print(building_heights[key_to_check])
except KeyError:
  print("That key doesn't exist!")
```

When we try to access a key that doesn't exist, the program will go into the `except` block and print `"That key doesn't exist!"`.

Example

```
caffeine_level = {"espresso": 64, "chai": 40, "decaf": 0, "drip": 120}
# caffeine_level["matcha"] = 30


try:
  caffeine_level["matcha"] = 30
  print(caffeine_level)
except KeyError:
  print("Unknown Caffeine Level")
```

Safely Get a Key

We saw in the last exercise that we had to add a key:value pair to a dictionary in order to avoid a KeyError. This solution is not sustainable. We can't predict every key a user may call and add all of those placeholder values to our dictionary!

Dictionaries have a `.get()` method to search for a value instead of the `my_dict[key]` notation we have been using. If the key you are trying to `.get()` does not exist, it will return `None` by default:

```
building_heights = {"Burj Khalifa": 828, "Shanghai Tower": 632,
"Abraj Al Bait": 601, "Ping An": 599, "Lotte World Tower": 554.5,
"One World Trade": 541.3}

#this line will return 632:
building_heights.get("Shanghai Tower")

#this line will return None:
building_heights.get("My House")
```

You can also specify a value to return if the key doesn't exist. For example, we might want to return a building height of 0 if our desired building is not in the dictionary:

```
>>> building_heights.get('Shanghai Tower', 0)
632
>>> building_heights.get('Mt Olympus', 0)
0
>>> building_heights.get('Kilimanjaro', 'No Value')
'No Value'
```

Example

```
user_ids = {"teraCoder": 100019, "pythonGuy": 182921, "samTheJavaMaam": 123112
 "lyleLoop": 102931, "keysmithKeith": 129384}

tc_id = user_ids.get("teraCoder", 100000)
print(tc_id)

stack_id = user_ids.get("superStackSmash", 100000)
print(stack_id)
```

Delete a Key

Sometimes we want to get a key and remove it from the dictionary. Imagine we were running a raffle, and we have this dictionary mapping ticket numbers to prizes:

```
raffle = {223842: "Teddy Bear", 872921: "Concert Tickets", 320291:
"Gift Basket", 412123: "Necklace", 298787: "Pasta Maker"}
```

When we get a ticket number, we want to return the prize and also remove that pair from the dictionary, since the prize has been given away. We can use `.pop()` to do this. Just like with `.get()`, we can provide a default value to return if the key does not exist in the dictionary:

```
>>> raffle.pop(320291, "No Prize")
"Gift Basket"
>>> raffle
{223842: "Teddy Bear", 872921: "Concert Tickets", 412123: "Necklace"
298787: "Pasta Maker"}
>>> raffle.pop(100000, "No Prize")
"No Prize"
>>> raffle
{223842: "Teddy Bear", 872921: "Concert Tickets", 412123: "Necklace"
298787: "Pasta Maker"}
>>> raffle.pop(872921, "No Prize")
"Concert Tickets"
>>> raffle
{223842: "Teddy Bear", 412123: "Necklace", 298787: "Pasta Maker"}
```

`.pop()` works to delete items from a dictionary, when you know the key value.

Example

```
available_items = {"health potion": 10, "cake of the cure": 5, "green elixir":
 20, "strength sandwich": 25, "stamina grains": 15, "power stew": 30}
health_points = 20 + available_items.pop("stamina grains", 0)
health_points = available_items.pop("power stew", 0) + health_points
health_points = available_items.pop("mystic bread", 0) + health_points

print(available_items, health_points)
```

Get All Keys

Sometimes we want to operate on all of the keys in a dictionary. For example, if we have a dictionary of students in a math class and their grades:

```
test_scores = {"Grace":[80, 72, 90], "Jeffrey":[88, 68, 81],
"Sylvia":[80, 82, 84], "Pedro":[98, 96, 95], "Martin":[78, 80, 78],
"Dina":[64, 60, 75]}
```

We want to get a roster of the students in the class, without including their grades. We can do this with the built-in `list()` function:

```
>>> list(test_scores)
["Grace", "Jeffrey", "Sylvia", "Pedro", "Martin", "Dina"]
```

Dictionaries also have a `.keys()` method that returns a `dict_keys` object.
A `dict_keys` object is a *view* object, which provides a look at the current state of the dictionary, without the user being able to modify anything. The `dict_keys` object returned by `.keys()` is a set of the keys in the dictionary. You cannot add or remove elements from a `dict_keys` object, but it can be used in the place of a list for iteration:

```
for student in test_scores.keys():
  print(student)
```

will yield:

```
Grace
Jeffrey
Sylvia
Pedro
Martin
Dina
```

Example

```
user_ids = {"teraCoder": 100019, "pythonGuy": 182921, "samTheJavaMaam": 123112
 "lyleLoop": 102931, "keysmithKeith": 129384}
num_exercises = {"functions": 10, "syntax": 13, "control flow": 15, "loops": 2
2, "lists": 19, "classes": 18, "dictionaries": 18}

users = user_ids.keys()
lessons = num_exercises.keys()

print(users)
print(lessons)
```

Get All Values

Dictionaries have a `.values()` method that returns a `dict_values` object (just like a `dict_keys` object but for values!) with all of the values in the dictionary. It can be used in the place of a list for iteration:

```
test_scores = {"Grace":[80, 72, 90], "Jeffrey":[88, 68, 81],
"Sylvia":[80, 82, 84], "Pedro":[98, 96, 95], "Martin":[78, 80, 78],
"Dina":[64, 60, 75]}

for score_list in test_scores.values():
  print(score_list)
```

will yield:

```
[80, 72, 90]
[88, 68, 81]
[80, 82, 84]
[98, 96, 95]
[78, 80, 78]
[64, 60, 75]
```
There is no built-in function to get all of the values as a list, but if you *really* want to, you can use:

```
list(test_scores.values())
```
However, for most purposes, the `dict_values` object will act the way you want a list to act.

Example

```
num_exercises = {"functions": 10, "syntax": 13, "control flow": 15, "loops": 22, "lists": 19, "classes": 18, "dictionaries": 18}
total_exercises = 0
for x in num_exercises.values():
  total_exercises += x
print(total_exercises)
```

Get All Items

You can get both the keys and the values with the `.items()` method. Like `.keys()` and `.values()`, it returns a `dict_list` object. Each element of the `dict_list` returned by `.items()` is a tuple consisting of:

```
(key, value)
```
so to iterate through, you can use this syntax:

```
biggest_brands = {"Apple": 184, "Google": 141.7, "Microsoft": 80, "Coca-Cola": 69.7, "Amazon": 64.8}

for company, value in biggest_brands.items():
  print(company + " has a value of " + str(value) + " billion dollars. ")
```
which would yield this output:

```
Apple has a value of 184 billion dollars.
Google has a value of 141.7 billion dollars.
Microsoft has a value of 80 billion dollars.
Coca-Cola has a value of 69.7 billion dollars.
Amazon has a value of 64.8 billion dollars.
```

Example

```
pct_women_in_occupation = {"CEO": 28, "Engineering Manager": 9, "Pharmacist":
58, "Physician": 40, "Lawyer": 37, "Aerospace Engineer": 9}

for women, value in pct_women_in_occupation.items():
  print(f"Women make up {value} percent of {women}s")
```