# Creating Dictionaries

About 30 minutes

Welcome to your next course — Dictionaries in Python!

This first introductory lesson will teach you how to create a dictionary, a new powerful datatype. By the end of the lesson, you will know how to make different kinds of dictionaries and understand what they represent.

# What is a Dictionary?

A dictionary is an unordered set of key: value pairs.

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

* Oatmeal is 3 dollars
* 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 = {"oatmeal": 3, "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 (i.e., "oatmeal") and a value (i.e., 3)
3. Each key: value pair (i.e., "oatmeal": 3 or "avocado toast": 6) is separated by a comma (,)
4. It's considered good practice to insert a space ( ) after each comma, but your code will still run without the space.

Dictionaries provide 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.

**1.**

You have a dictionary of temperature sensors in your house and what temperatures they read. You've just added a sensor to your "pantry", and it reads 22 degrees. Add this pair to the dictionary on line 1.

**2.**

Remove the # in front of the definition of the dictionary num\_cameras, which represents the number of cameras in each area around your house. If you run this code, you'll get an error:

SyntaxError: invalid syntax

Try to find and fix the syntax error to make this code run.

**Hint:**

Add commas (,) to num\_cameras so that it runs without errors.

**Solution:**

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., "oatmeal": 3). 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 any type. You 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.

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

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

**1.**

Create a dictionary called translations that maps the following words in English to their definitions in Sindarin (the language of the elves):

| **English** | **Sindarin** |
| --- | --- |
| mountain | orod |
| bread | bass |
| friend | mellon |
| horse | roch |

**Solution:**

translations = {"mountain": "orod", "bread": "bass", "friend": "mellon", "horse": "roch"}

# 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}

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.

**1.**

Run the code. You should get an error:

TypeError: unhashable type

Make the code run without errors by flipping the items in the dictionary so that the strings are the keys and the lists are the values

**Solution:**

children = {"von Trapp": ["Johannes", "Rosmarie", "Eleonore"],"Corleone": ["Sonny", "Fredo", "Michael"]}

# Empty Dictionary

A dictionary doesn't have to contain anything. You can create an empty dictionary:

empty\_dict = {}

We can create an empty dictionary when we plan to fill it later based on some other input. We will explore ways to fill a dictionary in the next exercise.

Instructions

**1.**

Create an empty dictionary called my\_empty\_dictionary.

**Solution:**

my\_empty\_dictionary = {}

# Add A Key

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

my\_dict["new\_key"] = "new\_value"

For example, if we had our menu object 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}

**1.**

Create an empty dictionary called animals\_in\_zoo.

**2.**

Walking around the zoo, you see 8 zebras. Add "zebras" to animals\_in\_zoo as a key with a value of 8.

**3.**

The primate house was bananas! Add "monkeys" to animals\_in\_zoo as a key with a value of 12.

**4.**

As you leave the zoo, you are saddened that you did not see any dinosaurs. Add "dinosaurs" to animals\_in\_zoo as a key with a value of 0.

**5.**

Print animals\_in\_zoo.

**Solution**:

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 the first 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})

which 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}

Instructions

**1.**

In one line of code, add two new users to the user\_ids dictionary:

* theLooper, with an id of 138475
* stringQueen, with an id of 85739

**2.**

Print user\_ids.

**Solution:**

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

which 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)

would yield:

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

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

**1.**

Add the key "Supporting Actress" and set the value to "Viola Davis".

**2.**

Without changing the definition of the dictionary oscar\_winners, change the value associated with the key "Best Picture" to "Moonlight".

**Solution:**

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'

# List Comprehensions to Dictionaries

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 list 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 a list of pairs. This list comprehension:

1. Takes a pair from the zipped list of pairs from names and heights
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 list of pairs

**1.**

You have two lists, representing some drinks sold at a coffee shop and the milligrams of caffeine in each. First, create a variable called zipped\_drinks that is a list of pairs between the drinks list and the caffeine list.

**2.**

Create a dictionary called drinks\_to\_caffeine by using a list comprehension that goes through the zipped\_drinks list and turns each pair into a key:value item.

**Solution:**

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)

# Review

So far we have learned:

* How to create a dictionary
* How to add elements to a dictionary
* How to update elements in a dictionary
* How to use a list comprehension to create a dictionary from two lists

Let's practice these skills!

**1.**

We are building a music streaming service. We have provided two lists, representing songs in a user's library and the amount of times each song has been played.

Using a list comprehension, create a dictionary called plays that goes through zip(songs, playcounts) and creates a song:playcount pair for each song in songs and each playcount in playcounts.

**2.**

Print plays.

**3.**

After printing plays, add a new entry to it. The entry should be for the song "Purple Haze" and the playcount is 1.

**4.**

This user has caught Aretha Franklin fever and listened to "Respect" 5 more times. Update the value for "Respect" to be 94 in the plays dictionary.

**5.**

Create a dictionary called library that has two key: value pairs:

* key "The Best Songs" with a value of plays, the dictionary you created
* key "Sunday Feelings" with a value of an empty dictionary

**6.**

Print library.

**Solution:**

songs = ["Like a Rolling Stone", "Satisfaction", "Imagine", "What's Going On", "Respect", "Good Vibrations"]

playcounts = [78, 29, 44, 21, 89, 5]

plays = {song:playcount for [song, playcount] in zip(songs, playcounts)}

plays["Respect"] = 94

plays["Purple Haze"] = 1

library = {"The Best Songs": plays, "Sunday Feelings": {}}

print(library)

# Using Dictionaries

About 40 minutes

This second lesson on dictionaries in Python will teach you how to use dictionaries. By the end of the lesson, you will know how to access different items in a dictionary and perform operations on them. You will see many different examples of dictionaries and solidify your understanding of this essential data structure.

# 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
* Find the length of a dictionary
* 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

**1.**

We have provided a dictionary that maps the elements of astrology to the zodiac signs. Print out the list of zodiac signs associated with the "earth" element.

**2.**

Print out the list of the "fire" signs.

**Solution:**

zodiac\_elements = {"water": ["Cancer", "Scorpio", "Pisces"], "fire": ["Aries", "Leo", "Sagittarius"], "earth": ["Taurus", "Virgo", "Capricorn"], "air":["Gemini", "Libra", "Aquarius"]}

print(zodiac\_elements['earth'])

print(zodiac\_elements['fire'])

# 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.

**1.**

Run the code. It should throw a KeyError! "energy" does not exist as one of the elements.

**2.**

Add the key "energy" to the zodiac\_elements. It should map to a value of "Not a Zodiac element". Run the code. Did this resolve the KeyError?

**Solution:**

zodiac\_elements = {"water": ["Cancer", "Scorpio", "Pisces"], "fire": ["Aries", "Leo", "Sagittarius"], "earth": ["Taurus", "Virgo", "Capricorn"], "air":["Gemini", "Libra", "Aquarius"]}

zodiac\_elements['energy'] = 'Not a Zodiac element'

print(zodiac\_elements["energy"])

# Try/Except to Get a Key

We saw that we can avoid KeyErrors 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!".

**1.**

Use a try block to try to print the caffeine level of "matcha". If there is a KeyError, print "Unknown Caffeine Level".

**2.**

Above the try block, add "matcha" to the dictionary with a value of 30.

caffeine\_level = {"espresso": 64, "chai": 40, "decaf": 0, "drip": 120}

caffeine\_level['matcha'] = 30

key\_to\_check = "matcha"

try:

print(caffeine\_level[key\_to\_check])

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'

**1.**

Use .get() to get the value of "teraCoder"'s user ID, with 100000 as a default value if the user doesn't exist. Store it in a variable called tc\_id. Print tc\_id to the console.

**2.**

Use .get() to get the value of "superStackSmash"'s user ID, with 100000 as a default value if the user doesn't exist. Store it in a variable called stack\_id. Print stack\_id to the console.

**Solution:**

user\_ids = {"teraCoder": 100019, "pythonGuy": 182921, "samTheJavaMaam": 123112, "lyleLoop": 102931, "keysmithKeith": 129384}

tc\_id = user\_ids.get("teraCoder", 100000)

stack\_id = user\_ids.get("superStackSmash", 100000)

print(tc\_id)

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.

**1.**

You are designing the video game Big Rock Adventure. We have provided a dictionary of items in the player's inventory to add points to their health meter. In one line, add the value of "stamina grains" to health\_points and remove the item from the dictionary. If the key does not exist, add 0 to health\_points.

**2.**

In one line, add the value of "power stew" to health\_points and remove the item from the dictionary. If the key does not exist, add 0 to health\_points.

**3.**

In one line, add the value of "mystic bread" to health\_points and remove the item from the dictionary. If the key does not exist, add 0 to health\_points.

**4.**

Print available\_items and health\_points.

**Solution:**

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

health\_points += available\_items.pop("stamina grains", 0)

health\_points += available\_items.pop("power stew", 0)

health\_points += available\_items.pop("mystic bread", 0)

print(available\_items)

print(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 dicitonary, 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"

**1.**

Create a variable called users and assign it to be all of the keys of the user\_ids list.

**2.**

Create a variable called lessons and assign it to be all of the keys of the num\_exercises list.

**3.**

Print users to the console.

**4.**

Print lessons to the console.

**Solution:**

user\_ids = {"teraCoder": 100019, "pythonGuy": 182921, "samTheJavaMaam": 123112, "lyleLoop": 102931, "keysmithKeith": 129384}

num\_exercises = {"functions": 10, "syntax": 13, "control flow": 15, "loops": 22, "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\_list object will act the way you want a list to act.

**1.**

Create a variable called total\_exercises and set it equal to 0.

**2.**

Iterate through the values in the num\_exercises list and add each value to the total\_exercises variable.

**3.**

Print the total\_exercises variable to the console.

**Solution:**

num\_exercises = {"functions": 10, "syntax": 13, "control flow": 15, "loops": 22, "lists": 19, "classes": 18, "dictionaries": 18}

total\_exercises = 0

for values in num\_exercises.values():

total\_exercises += values

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.

**1.**

Use a for loop to iterate through the items of pct\_women\_in\_occupation. For each key : value pair, print out a string that looks like:

Women make up [value] percent of [key]s.

**Solution:**

pct\_women\_in\_occupation = {"CEO": 28, "Engineering Managers": 9, "Pharmacist": 58, "Physician": 40, "Lawyer": 37, "Aerospace Engineer": 9}

for job, count in pct\_women\_in\_occupation.items():

print("Women make up " + str(count) + " percent of " + job)

# Review

In this lesson, you've learned how to go through dictionaries and access keys and values in different ways. Specifically you have seen how to:

* Use a key to get a value from a dictionary
* Check for existence of keys
* Find the length of a dictionary
* Remove a key: value pair from a dictionary
* Iterate through keys and values in dictionaries

**1.**

We have provided a pack of tarot cards, tarot. You are going to do a three card spread of your past, present, and future.

Create an empty dictionary called spread.

**2.**

The first card you draw is card 13. Pop the value assigned to the key 13 out of the tarot dictionary and assign it as the value of the "past" key of spread.

Hint:

To pop a value from one dictionary and assign it to another, you can use syntax like:

new\_dict["new key"] = old\_dict.pop("old key")

**3.**

The second card you draw is card 22. Pop the value assigned to the key 22 out of the tarot dictionary and assign it as the value of the "present" key of spread.

**4.**

The third card you draw is card 10. Pop the value assigned to the key 10 out of the tarot dictionary and assign it as the value of the "future" key of spread.

**5.**

Iterate through the items in the spread dictionary and for each key: value pair, print out a string that says:

Your {key} is the {value} card.

**6.**

Congratulations! You have learned about how to modify and use dictionaries.

**Solution:**

tarot = { 1: "The Magician", 2: "The High Priestess", 3: "The Empress", 4: "The Emperor", 5: "The Hierophant", 6: "The Lovers", 7: "The Chariot", 8: "Strength", 9: "The Hermit", 10: "Wheel of Fortune", 11: "Justice", 12: "The Hanged Man", 13: "Death", 14: "Temperance", 15: "The Devil", 16: "The Tower", 17: "The Star", 18: "The Moon", 19: "The Sun", 20: "Judgement", 21: "The World", 22: "The Fool"}

spread = {}

spread["past"] = tarot.pop(13)

spread["present"] = tarot.pop(22)

spread["future"] = tarot.pop(10)

for key, value in spread.items():

print("Your "+key+" is the "+value+" card. ")

# Project Scrabble

About 45 minutes

In this project, you will create some functions to score a Scrabble game. To do this, you will need to understand how to create and use dictionaries. On the way, you'll refresh your knowledge of loops and some string methods. If you get stuck or confused, remember that your Slack community is there to help!

This project is not graded, and you do not need to submit it anywhere. If you would like to check your results, the [solution code can be found here.](https://s3.amazonaws.com/codecademy-content/programs/programming-with-python/On-platform+solutions/scrabble.py)

# Scrabble

In this project, you will process some data from a group of friends playing scrabble. You will use dictionaries to organize players, words, and points.

There are many ways you can extend this project on your own if you finish and want to get more practice!

Learn: Python Dictionaries

# Scrabble

In this project, you will process some data from a group of friends playing scrabble. You will use dictionaries to organize players, words, and points.

There are many ways you can extend this project on your own if you finish and want to get more practice!

Tasks

0/15Complete

Mark the tasks as complete by checking them off

### Build your Point Dictionary

1.

We have provided you with two lists, letters and points. We would like to combine these two into a dictionary that would map a letter to its point value.

Using a list comprehension and zip, create a dictionary called letter\_to\_points that has the elements of letters as the keys and the elements of points as the values.

Remember that the syntax for creating a dictionary using a list comprehension is:

{key:value for key, value in zip(keys, values)}

In this case, the letters list represents keys and the points list represents values.

2.

Our letters list did not take into account blank tiles. Add an element to the letter\_to\_points dictionary that has a key of " " and a point value of 0.

To add an element to a dictionary, you can use the syntax:

my\_dict[key\_to\_add] = value\_of\_key

### Score a Word

3.

We want to create a function that will take in a word and return how many points that word is worth.

Define a function called score\_word that takes in a parameter word.

Remember that to define a function, we can use the syntax:

def my\_function(my\_parameter):

... code execution here...

4.

Inside score\_word, create a variable called point\_total and set it to 0.

5.

After defining point\_total, create a for loop that goes through the letters in word and adds the point value of each letter to point\_total.

You should get the point value from the letter\_to\_points dictionary. If the letter you are checking for is not in letter\_to\_points, add 0 to the point\_total.

To iterate through each character in a string, you can use the syntax:

for char in long\_string:

... do something with char...

To get a value from a dictionary, with a default return of 0, you can use:

my\_dict.get(key\_to\_get, 0)

6.

After the for loop is finished, return point\_total.

7.

Let's test this function! Create a variable called brownie\_points and set it equal to the value returned by the score\_words() function with an input of "BROWNIE".

8.

We expect the word BROWNIE to earn 15 points:

(B + R + O + W + N + I + E)

(3 + 1 + 1 + 4 + 4 + 1 + 1) = 15

Let's print out brownie\_points to make sure we got it right.

### Score a Game

9.

Create a dictionary called player\_to\_words that maps players to a list of the words they have played. This table represents the data to transcribe into your dictionary:

| **player1** | **wordNerd** | **Lexi Con** | **Prof Reader** |
| --- | --- | --- | --- |
| BLUE | EARTH | ERASER | ZAP |
| TENNIS | EYES | BELLY | COMA |
| EXIT | MACHINE | HUSKY | PERIOD |

A dictionary with one player, Player Lonely, who has played the three words WISHING, FOR, and FRIENDS, would look like:

player\_to\_words = {"Player Lonely": ["WISHING", "FOR", "FRIENDS"]}

10.

Create an empty dictionary called player\_to\_points.

To initialize an empty dictionary, use empty brackets ({}).

11.

Iterate through the items in player\_to\_words. Call each player player and each list of words words.

Within your loop, create a variable called player\_points and set it to 0.

To iterate through items in a dictionary, we can use this syntax:

for key, value in my\_dict.items():

... do something with key or value ...

12.

Within the loop, create another loop that goes through each word in words and adds the value of score\_word() with word as an input.

Nested loops look like this:

for item1 in one\_list\_or\_dict:

for item2 in another\_object:

... do something with item1 or item2 ...

13.

After the inner loop ends, set the current player value to be a key of player\_to\_points, with a value of player\_points.

To set a key:value pair in a dictionary, use the syntax:

my\_dict[key\_to\_add] = value\_to\_add

14.

player\_to\_points should now contain the mapping of players to how many points they've scored. Print this out to see the current standings for this game!

If you've calculated correctly, wordNerd should be winning by 1 point.

### Ideas for Further Practice!

15.

If you want extended practice, try to implement some of these ideas with the Python you've learned:

* play\_word() — a function that would take in a player and a word, and add that word to the list of words they've played
* update\_point\_totals() — turn your nested loops into a function that you can call any time a word is played
* make your letter\_to\_points dictionary able to handle lowercase inputs as well

**Solution:**

letters = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"]

points = [1, 3, 3, 2, 1, 4, 2, 4, 1, 8, 5, 1, 3, 4, 1, 3, 10, 1, 1, 1, 1, 4, 4, 8, 4, 10]

#learner code here:

letters\_to\_points = {key:value for key, value in zip(letters,points)}

letters\_to\_points[" "] = 0

def score\_word(word):

point\_total = 0

for letter in word:

if letter in letters\_to\_points:

point\_total += letters\_to\_points[letter]

return point\_total

brownie\_points = score\_word("Brownie")

print(brownie\_points)

players\_to\_words = {"player1":["BLUE", "TENNIS", "EXIT"], "wordNerd":["EARTH", "EYES", "MACHINE"], "Lexi Con": ["ERASER", "BELLY", "HUSKY"], "Prof Reader": ["ZAP", "COMA", "PERIOD"]}

player\_to\_points = {}

for player, words in players\_to\_words.items():

score = 0

for word in words:

score += score\_word(word)

player\_to\_points[player] = score

print(player\_to\_points)

# Dictionaries - Code Challenge

Now you have learned about the basics of Python syntax, control flow, lists, loops, dictionaries, string methods and how to create functions that run repeatable blocks of code.

Let's reinforce these concepts with a series of practice problems.

# Introduction

This lesson will help you review Python functions by providing some challenge exercises involving dictionaries.

As a refresher, function syntax looks like this:

def some\_function(some\_input1, some\_input2):

… do something with the inputs …

return output

For example, a function that counts the number of values in a dictionary that are above a given number would look like this:

def greater\_than\_ten(my\_dictionary, number):

count = 0

for value in my\_dictionary.values():

if value > number:

count += 1

return count

And this would produce output like:

>>> greater\_than\_ten({"a":1, "b":2, "c":3}, 0)

3

>>> greater\_than\_ten({"a":1, "b":2, "c":3}, 5)

0

Instructions

When you're ready to do this series of short function challenges, continue on to the rest of the lesson!

# Sum Values

sum\_values()

Instructions

**1.**

Write a function named sum\_values that takes a dictionary named my\_dictionary as a parameter. The function should return the sum of the values of the dictionary

Create a counter variable and start it at 0. Loop through all of the elements of my\_dictionary.values() and add each value to your counter variable.

**Solution:**

# Write your sum\_values function here:

def sum\_values(my\_dictionary):

total = 0

for value in my\_dictionary.values():

total += value

return total

# Uncomment these function calls to test your sum\_values function:

print(sum\_values({"milk":5, "eggs":2, "flour": 3}))

# should print 10

print(sum\_values({10:1, 100:2, 1000:3}))

# should print 6

# Even Keys

sum\_even\_keys()

Instructions

**1.**

Create a function called sum\_even\_keys that takes a dictionary named my\_dictionary, with all integer keys and values, as a parameter. This function should return the sum of the values of all even keys.

Create a counter variable and start it at 0. Loop through all of the elements of the keys of the dictionary by using my\_dictionary.keys(). If the key is even (which you can check by using key % 2 == 0), add the corresponding value to the counter.

**Solution:**

# Write your sum\_even\_keys function here:

def sum\_even\_keys(my\_dictionary):

total = 0

for key in my\_dictionary.keys():

if key%2 == 0:

total += my\_dictionary[key]

return total

# Uncomment these function calls to test your function:

print(sum\_even\_keys({1:5, 2:2, 3:3}))

# should print 2

print(sum\_even\_keys({10:1, 100:2, 1000:3}))

# should print 6

# Add Ten

add\_ten()

Instructions

**1.**

Create a function named add\_ten that takes a dictionary with integer values named my\_dictionary as a parameter. The function should add 10 to every value in my\_dictionary and return my\_dictionary

**Solution:**

# Write your add\_ten function here:

def add\_ten(my\_dictionary):

for key in my\_dictionary.keys():

my\_dictionary[key] += 10

return my\_dictionary

# Uncomment these function calls to test your function:

print(add\_ten({1:5, 2:2, 3:3}))

# should print {1:15, 2:12, 3:13}

print(add\_ten({10:1, 100:2, 1000:3}))

# should print {10:11, 100:12, 1000:13}

# Values That Are Keys

values\_that\_are\_keys()

Instructions

**1.**

Create a function named values\_that\_are\_keys that takes a dictionary named my\_dictionary as a parameter. This function should return a list of all values in the dictionary that are also keys.

Loop through all values in the dictionary by using for value in my\_dictionary.values(). Check to see if value is in my\_dictionary.keys() and if so, append it to a list.

**Solution:**

# Write your values\_that\_are\_keys function here:

def values\_that\_are\_keys(my\_dictionary):

value\_keys = []

for value in my\_dictionary.values():

if value in my\_dictionary:

value\_keys.append(value)

return value\_keys

# Uncomment these function calls to test your function:

print(values\_that\_are\_keys({1:100, 2:1, 3:4, 4:10}))

# should print [1, 4]

print(values\_that\_are\_keys({"a":"apple", "b":"a", "c":100}))

# should print ["a"]

**Largest Value**

max\_key()

Instructions

**1.**

Write a function named max\_key that takes a dictionary named my\_dictionary as a parameter. The function should return the key associated with the largest value in the dictionary.

Begin by creating two variables named largest\_key and largest\_value. Initialize largest\_value to be the smallest number possible (you can use float("-inf"). Initialize largest\_key to be an empty string.

Loop through all keys/value pair in the dictionary. Any time you find a value larger than what is currently stored in largest\_value, replace largest\_value with that new value. Similarly, replace largest\_key with the key associated with the new largest value.

After looping through all key/value pairs, return largest\_key.

**Solution:**

# Write your max\_key function here:

def max\_key(my\_dictionary):

largest\_key = float("-inf")

largest\_value = float("-inf")

for key, value in my\_dictionary.items():

if value > largest\_value:

largest\_value = value

largest\_key = key

return largest\_key

# Uncomment these function calls to test your function:

print(max\_key({1:100, 2:1, 3:4, 4:10}))

# should print 1

print(max\_key({"a":100, "b":10, "c":1000}))

# should print "c"

**Word Length Dict**

word\_length\_dictionary()

Instructions

**1.**

Write a function named word\_length\_dictionary that takes a list of strings named words as a parameter. The function should return a dictionary of key/value pairs where every key is a word in words and every value is the length of that word.

First create an empty dictionary named something like word\_lengths. Loop through every word in words and add a new key using word\_lengths[word] = len(word)

**Solution**:

# Write your word\_length\_dictionary function here:

def word\_length\_dictionary(words):

word\_lengths = {}

for word in words:

word\_lengths[word] = len(word)

return word\_lengths

# Uncomment these function calls to test your function:

print(word\_length\_dictionary(["apple", "dog", "cat"]))

# should print {"apple":5, "dog": 3, "cat":3}

print(word\_length\_dictionary(["a", ""]))

# should print {"a": 1, "": 0}

# Frequency Count

frequency\_dictionary()

Instructions

**1.**

Write a function named frequency\_dictionary that takes a list of elements named words as a parameter. The function should return a dictionary containing the frequency of each element in words.

First, create a new empty dictionary. Then, loop through every word in words. If word is not a key in the dictionary, make word a key with a value of 1. If word was already a key, increase the value associated with word by 1.

**Solution:**

# Write your frequency\_dictionary function here:

def frequency\_dictionary(words):

freqs = {}

for word in words:

if word not in freqs:

freqs[word] = 0

freqs[word] += 1

return freqs

# Uncomment these function calls to test your function:

print(frequency\_dictionary(["apple", "apple", "cat", 1]))

# should print {"apple":2, "cat":1, 1:1}

print(frequency\_dictionary([0,0,0,0,0]))

# should print {0:5}

# Unique Values

Begin by creating a new empty list named seen\_values. Loop through all of the values of my\_dictionary. For every value, check to see if that value is in seen\_values. If it is, continue to the next value. If it is not, add it to seen\_values. After looping through all values, return the length of seen\_values.

Begin by creating a new empty list named seen\_values. Loop through all of the values of my\_dictionary. For every value, check to see if that value is in seen\_values. If it is, continue to the next value. If it is not, add it to seen\_values. After looping through all values, return the length of seen\_values.

**Solution:**

# Write your unique\_values function here:

def unique\_values(my\_dictionary):

seen\_values = []

for value in my\_dictionary.values():

if value not in seen\_values:

seen\_values.append(value)

return len(seen\_values)

# Uncomment these function calls to test your function:

print(unique\_values({0:3, 1:1, 4:1, 5:3}))

# should print 2

print(unique\_values({0:3, 1:3, 4:3, 5:3}))

# should print 1

# Count First Letter

count\_first\_letter()

Instructions

**1.**

Create a function named count\_first\_letter that takes a dictionary named names as a parameter. names should be a dictionary where the key is a last name and the value is a list of first names. For example, the dictionary might look like this:

names = {"Stark": ["Ned", "Robb", "Sansa"], "Snow" : ["Jon"], "Lannister": ["Jaime", "Cersei", "Tywin"]}

The function should return a new dictionary where each key is the first letter of a last name, and the value is the number of people whose last name begins with that letter.

So in example above, the function would return:

{"S" : 4, "L": 3}

Begin by creating an empty dictionary named something like letters. Loop through the keys of names and access the first letter of each the key using key[0].

If that letter is not a key in letters, create a new key/value pair where the key is key[0] and the value is the length of names[key].

If that letter is a key in letters, simply add the length of names[key] to value associated with key[0] in letters.

**Solution:**

# Write your count\_first\_letter function here:

def count\_first\_letter(names):

letters = {}

for key in names:

first\_letter = key[0]

if first\_letter not in letters:

letters[first\_letter] = 0

letters[first\_letter] += len(names[key])

return letters

# Uncomment these function calls to test your function:

print(count\_first\_letter({"Stark": ["Ned", "Robb", "Sansa"], "Snow" : ["Jon"], "Lannister": ["Jaime", "Cersei", "Tywin"]}))

# should print {"S": 4, "L": 3}

print(count\_first\_letter({"Stark": ["Ned", "Robb", "Sansa"], "Snow" : ["Jon"], "Sannister": ["Jaime", "Cersei", "Tywin"]}))

# should print {"S": 7}

# Modules

About 8 minutes

## Introduction

In the world of programming, a great deal of emphasis is placed on making code reusable. It's a fair question to ask — reusable by whom? In most cases, we write code in a way that it can be reusable by ourselves. But, when a frequently-encountered problem is solved in a general way that can prove useful to a broader audience, we might hope that our code can be reusable in a much larger sense. In this article, we're going to explore how to use tools built in Python that are not built into the programming language.

Python allows us to package code into files or sets of files called modules. A module is a collection of Python declarations intended broadly to be used as a tool. Say you were writing software that handles monetary transactions. Being familiar with [rounding errors in floating-point arithmetic](https://docs.python.org/3/tutorial/floatingpoint.html) you want to use a data type that performs decimal arithmetic more accurately. You could do the following:

from decimal import Decimal

cost\_of\_gum = Decimal('0.10')

cost\_of\_gumdrop = Decimal('0.35')

cost\_of\_transaction = cost\_of\_gum + cost\_of\_gumdrop

# Returns Decimal('0.45') instead of 0.44999999999999996

Above, we use the decimal module's Decimal data type to add .10 with .35. If we used floating-point arithmetic this sum would result in a weirdly formatted number. Since we used the Decimal type the arithmetic acts much more as expected.

Usually, modules will provide functions or data types that we can then use to solve a general problem, allowing us more time to focus on the software that we are building to solve a more specific problem.

#### Importing Modules

Since a module is, by definition, code outside of the file we're currently running, we need to tell Python to include that code before using the functions it offers. We do this using the import statement. This import statement tells Python to take whatever code is associated with a set of files and run them before running the rest of a script.

import math

In this code, we tell Python to import a library called math. As a result, it runs a set of files with various Python function definitions related to mathematical operations and allows us to use those functions in the code that we write.

Imagine we needed to calculate the sine of a number within a function. We could do that with the following:

import math

number = 10

sine\_of\_number = math.sin(number)

In this code, we first import the math library. After it is imported, we define a variable number to be the number we want to take the sine of. We then calculate the sine of number by calling the function math.sin() on it, saving the result in sine\_of\_number.

sin() is the name of one of the functions that math defines when it gets imported. The math module also includes many other functions, like sqrt() for finding a square root, log() for finding logarithms, and even constants like pi and e. A full list of the available functions in the math library is available in [the Python documentation](https://docs.python.org/3/library/math.html).

#### Namespaces

Notice that when we want to invoke the sin() function we call math.sin(). This is default behavior where Python offers a namespace for the module. A namespace isolates the functions, classes, and variables defined in the module from the code in the file doing the importing.

Python defaults to naming the namespace after the module being imported, but this name can be altered by aliasing using the as keyword. Aliasing is often done if the name of the library is long and typing the full name every time you want to use one of its functions is laborious.

import random as r

# Pick a random integer from 1 to 10

r.randint(1, 10)

Above, we imported the library random, but instead of using random as the namespace name we aliased the namespace to r. After that, we called a function from the random module called randint(), which returns a random integer between two given endpoints. Since we renamed the namespace, instead of calling random.randint() we called r.randint().

#### Importing into the local namespace

Sometimes importing a whole library is overkill when we only need a specific function or data type from that library. In this case, there's syntax we can use to import specific declarations from a module. The syntax is :

from {module} import {thing}

In order to import the date data type from the datetime library, for instance, we would do the following:

# imports the "date" data type from the "datetime" module

from datetime import date

# Python 3.0's release date

python3\_release = date(2008, 12, 3)

today = date.today()

days\_since\_python3\_release = today - python3\_release

Above, we import just the date part of the datetime module. Notice that we can just call date without using datetime.date() like we would have to if had just imported datetime. When using the from {module} import {thing} syntax, functions, classes, and variables are imported into the local namespace. The local namespace is where our code is being run.

#### DANGER: Namespace pollution

We can import all functions, classes, and variables from a library into the local namespace at once. The syntax to import everything looks like:

from {module} import \*

This syntax is considered dangerous because it could pollute our local namespace. Pollution occurs when the same name could apply to two possible things. For example:

from math import \*

def floor(tiles):

"""

Takes the number of tiles for a linoleum floor as an argument

Returns the price of that many tiles

"""

PRICE\_PER\_LINOLEUM\_TILES = 3.00

return tiles \* PRICE\_PER\_LINOLEUM\_TILES

customer1\_purchase\_price = floor(20)

# customer1\_purchase\_price is equal to 60.00

If we're writing software for a tiling company we may very well define a function called floor() that takes a number of linoleum floor tiles and returns the price for that many tiles. Say we decide to do some basic geometry as part of our services, so we import all of the math library into the local namespace.

It just so happens that the math module also implements a function called floor() which rounds a number down to the nearest integer. If we were to call floor() in a context where both our floor() and math's floor() are defined without a namespace, it would depend on whether our function definition or the import statement happened first, which can lead to unexpected behavior in our code.

Say we change floor(), admittedly not a very descriptive name, to floor\_tiling\_price(). If we are still importing math.floor() as floor() and change our floor() function's name to floor\_tiling\_price() code that calls the function floor() might run without exception but not be doing what we want at all.

from math import \*

def floor\_tiling\_price(tiles):

"""

Takes the number of tiles for a linoleum floor as an argument

Returns the price of that many tiles

"""

PRICE\_PER\_LINOLEUM\_TILES = 3.00

return tiles \* PRICE\_PER\_LINOLEUM\_TILES

customer1\_purchase\_price = floor(20)

# customer1\_purchase\_price is equal to 20

Above, we changed the name of our function floor() but we still import all of the math functions as before. We forgot to change customer1\_purchase\_price's calculation from floor() to floor\_tiling\_price(). Usually, this would cause a NameError and we'd be able to spot where we are miscalculating the price, but this would silently cause the wrong data to be saved into the customer's purchase. A costly error, indeed!

#### Wrapping Up

Programmers can do great things if they are not forced to constantly reinvent tools that have already been built. With the power of modules, we can import any code that someone else has shared publicly. In this article, we covered some of the Python Standard Library, but you can explore all the modules that come packaged with every installation of Python at the [Python Standard Library documentation](https://docs.python.org/3/library/index.html). This is just the beginning. Using a package manager (like conda or pip3), you can install any modules available on the [Python Package Index](https://pypi.org/search/).

The sky's the limit!

# Abruptly Goblins

About 2 minutes

In this project, off-platform project, you will be the proprietor of a comic and games store, the Sorcery Society, and you decide to host a game night to play the hot new tabletop RPG Abruptly Goblins! You'll be writing your own Python functions and using dictionaries to provide a new service to gaming attendees of your shop.

**This project is totally optional; if you’re busy this week, you can skip it!**

Your beloved clientele want to organize a game night at your comics store! Create a system for organizing and tracking gamer's availability. Programmatically figure out the best day to host a game night and send out emails to your attendees to let them know when to come. Follow the steps below to get started with your project!

#### Working on Your Computer

1. If you’ve never used the command line, we recommend taking the [Learn the Command Line course](https://www.codecademy.com/learn/learn-the-command-line).
2. Install Python by following the directions in this article on [Installing Python](https://www.codecademy.com/articles/install-python3).
3. Learn about [Jupyter Notebooks](https://www.codecademy.com/articles/how-to-use-jupyter-notebooks-py3), a cool way of combining Python code with explanations or instruction in a web terminal.
4. Download the [Abruptly Goblins project](https://s3.amazonaws.com/codecademy-content/programs/programming-with-python/Abruptly+Goblins.zip).
5. Unzip it by double-clicking on it.
6. In the terminal, navigate to the directory containing the project, and type:

jupyter notebook

This should open a browser tab.

1. Click on Abruptly Goblins Planner.ipynb in the browser tab. This will open up your Jupyter Notebook.
2. Follow the steps in the Jupyter Notebook. If you get stuck, you can look at Abruptly Goblins Planner (Solution).ipynb for the answer.