Introduction

In this video we're going to take a look at Python dictionaries. Dictionaries can store an almost limitless amount of information and are used in Python to connect pieces of related information. We're going to learn how to create dictionaries and then access and modify the information within them.

Python dictionaries provide us with the ability to create real-world objects more accurately within our code. We could create a dictionary that represents a car and then store as much information as we need about that car. We could store brand, model, specifications, price, and much more. Dictionaries will allow us to store any two kinds of information that can be matched up such as employees and their salaries, car brands and models and so on.

Our first dictionary

Let's get started by creating our first dictionary. Imagine you own a company with a single employee:

In [1]:
```
employee_001 = {'name' : 'tony staunton', 'salary' : 20000 }

print(employee_001['name'])
print(employee_001['salary'])
```

```
tony staunton
20000
```

Our first dictionary is created at line 1 using curly brackets {} and assigning it the name **employee_001**. The dictionary stores an employees names and their salary. At lines 3 and 4 we use a **print** statement to access the value of 'name' and 'salary' and display them on screen. This may all seem a bit abstract at the moment but with practice you will see the value of using dictionaries to model real-world scenarios.

Using dictionaries

In Python, a dictionary is made up of what are called **key-value** pairs, for example *'name':'tony staunton'*. Here, *name* is the key and *tony staunton* is the value. Each key is connected to a value and you can use a key to access the value associated with that key. In Python, you can use any object that you can create as a value. A key's value can be a number, a string, a list or another dictionary.

As we've just seen in the first example Python dictionaries are contained within curly brackets **{}**. Inside these brackets is a series of key-value pairs.

employee_001 = {'name' : 'tony staunton', 'salary' : 20000 }

A *key-value pair* is a set of values associated with each other. When you provide Python with a key, it returns the associated value. Every key is connected to its value by a colon

(:), and each key-value pair is separated from one another by a comma (,). You can store as many key-value pairs as you need in a dictionary.

Accessing values in a dictionary

In [2]:
```
employee_001 = {'name' : 'tony staunton', 'salary' : 20000 }

print(employee_001['salary'])
```

```
20000
```

This simple example returns the value associated with the key 'salary' from the dictionary _employee*001*.

Let's take a look at another example:

In [3]:
```
employee_001 = {'name' : 'tony staunton', 'salary' : 20000 }

current_salary = employee_001['salary']

print(f"Your current salary is ${current_salary}.")
```

```
Your current salary is $20000.
```

In this example we define a dictionary at line 1. Then at line 3 we create a variable called _current*salary* and assign to it the value associated with the key 'salary'. The *print* statement at line 5 informs the user of their current salary by displaying the variable _current*salary*.

Adding new key-value pairs

One of the great things about dictionaries is that they are dynamic, which means that you can add new key-value pairs at any time. Let's take a look at how to add additional key-value pairs to our _employee*001* dictionary:

In [4]:
```
employee_001 = {'name' : 'tony staunton', 'salary' : 20000 }
print(employee_001)

employee_001['grade'] = 'manager'
employee_001['time_of_service'] = 4

print(employee_001)
```

```
{'name': 'tony staunton', 'salary': 20000}
{'name': 'tony staunton', 'salary': 20000, 'grade': 'manager', 'time_of_serv
ice': 4}
```

In this code example we start as we have done previously by defining a new dictionary, _employee*001*. We then print out the dictionary at line 2 to confirm the information held within. At lines 4 and 5 we add new key-value pairs to the dictionary. These are:

- key 'grade' and value 'manager'
- key 'time_of_service' and value 4

At line 7 we print out the modified dictionary which now contains 4 key-value pairs.

> One of the most common questions from students who are working with dictionaries is related to the order in which they are output from a *print* statement. In Python 3.6 or earlier when you printed out the contents of a dictionary or looped through its elements you could see the elements in any order. As of Python 3.7, dictionaries retain the order in which they are defined.

Working with an empty dictionary

Imaging it is the first day of your new business. As of yet you have no employees. We can create an empty dictionary and then add employees to it as they join our business. To create an empty dictionary, we do so by using empty curly brackets, we can then add key-value pairs. Let's revisit our employee_001 dictionary:

In [5]:
```
employee_001 = {}

employee_001['name'] = 'jane smith'
employee_001['salary'] = 20000

print(employee_001)
```

```
{'name': 'jane smith', 'salary': 20000}
```

In this example we define an empty dictionary at line 1 and then add name and salary key-value pairs to it. At line 6 we print out the dictionary.

The best time to use an empty dictionary is when you are starting out with information that will be supplied by a user.

Changing values within a dictionary

As well as creating dictionaries, populated or empty, and adding key-value pairs to them we can also modify the values within them. To do this give the name of the dictionary with the key in square brackets and then the new value you want associated with that key. Imagine you have just been given a raise, let's take a look at how we can modify the value associated with the key 'salary':

In [7]:
```
employee_001 = {'name' : 'tony staunton', 'salary' : 20000 }
print(f"Your current salary is ${employee_001['salary']}.")

employee_001['salary'] = 30000
print(f"Your new salary is ${employee_001['salary']}.")
```

```
Your current salary is $20000.
Your new salary is $30000.
```

In this example we define a dictionary as we have done previously at line 1. The at line 4 we associate a new value '30000' with the key 'salary'. At line 5 we print out the new salary to show that the modification has been successful. Let's look at this example again but this time with a bit more added to it:

In [9]:
```
employee_001 = {'name':'tony staunton', 'salary':20000, 'time_of_service':12
print(f"Based on your current years of service, {employee_001['time_of_servi
```

```
# Increase salary based in years of service
if employee_001['time_of_service'] <= 1:
    salary_increment = 5000
elif employee_001['time_of_service'] <= 3:
    salary_increment = 10000
else:
    salary_increment = 15000

# The new salary is based on the current salary plus the salary increment.
employee_001['salary'] = employee_001['salary'] + salary_increment

print(f"Your new salary is: ${employee_001['salary']}")
```

```
Based on your current years of service, 12, your current salary is $20000.
Your new salary is: $35000
```

This is a great code example as it has a lot going on inside it. Let's walk-through it.

- At line 1 we define a new dictionary
- At line 2 we print out two sets of key-value pairs
- At line 5 we start an *if-elif-else* chain to determine what salary increment should be applied to the current salary
- If an employee's time of service is less than or equal to 1 year than a salary increment of 5000 is applied, if the time of service is less than or equal to 3 then then a salary increment of 10000 is applied, finally for everything greater than 3 a salary increment of 15000 is applied
- At line 13 we are replacing the current value of the key 'salary' with the result of the calculation 'salary' + salary_increment.
- At line 15 the new salary is displayed on screen

Removing key-value pairs

If you ever need to delete a piece of information that is stored in a dictionary you can use the *del* statement. This statement will completely remove a key-value pair, you cannot retrieve it later. To use the *del* statement you need to provide it with the dictionary name and the key to be removed. Let's take a look:

In [10]:
```
employee_001 = {'name':'tony staunton', 'salary':20000, 'time_of_service':1}
print(f"Based on your current years of service, {employee_001['time_of_servi

del employee_001['salary']
print(employee_001)
```

```
Based on your current years of service, 1, your current salary is $20000.
{'name': 'tony staunton', 'time_of_service': 1}
```

In this code example we use the *del* statement at line 4 to tell Python to delete the key 'salary' from the dictionary _employee*001*. This will also remove the value associate with the key as well.

Similar objects within a dictionary

So far our dictionaries have been storing different kinds of information about one object, an employee and their details. Dictionaries can also be used to store one kind of

information about many objects. Imagine you want to save your friends favorite food, a dictionary can used:

In [11]:
```
favorite_food = {'tony' : 'chicken wings',
                 'jane' : 'pasta',
                 'tom' : 'burgers',
                 'mary' : 'fish',
                 }
```

In the code above we have several keys, which are the names of friends and each value is their favorite food. As you can see this dictionary is defined over several lines with a line break after the comma. Let's see how we can use this dictionary:

In [14]:
```
favorite_food = {'tony' : 'chicken wings',
                 'jane' : 'pasta',
                 'tom' : 'burgers',
                 'mary' : 'fish',
                 }

food = favorite_food['mary'].title()
print(f"Mary's favorite food is {food}.")
```

```
Mary's favorite food is Fish.
```

This example should be mostly familiar to you by now. At line 7 we create a new variable called *food* and assigned to it the value which is associated with the key 'tom'. We then print that value out to the user. We can of course use this code to get the value of any key in the dictionary.

Access values with get()

What happens when you try to access a key in a dictionary that no longer exists or never existed? You'll get an error! Let's take a look:

In [15]:
```
employee_001 = {'name' : 'tony staunton'}
print(employee_001['salary'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Input In [15], in <cell line: 2>()
      1 employee_001 = {'name' : 'tony staunton'}
----> 2 print(employee_001['salary'])

KeyError: 'salary'
```

What we get here is called a traceback with a *KeyError*. To handle this error we use Python's *get()* method to set a default value that will be returned if the requested key does not exist.

To use the *get()* method we need to provide it with a key as a first argument. A second optional argument can be passed to the *get* method which should contain the value to be returned if the requested key does not exist. Let's look at the example above, this time including the *get* method:

In [16]:
```
employee_001 = {'name' : 'tony staunton'}
```

```
current_salary = employee_001.get('salary', 'No salary is assigned.')

print(current_salary)
```

```
No salary is assigned.
```

Consider using the *get()* method when you think that a key that may be requested might not exist. What happens if we leave out the second optional argument of the *get* method?

In [17]:
```
employee_001 = {'name' : 'tony staunton'}

current_salary = employee_001.get('salary')

print(current_salary)
```

```
None
```

None is returned which means that no value exists.

Exercises

- Use a dictionary to store information about a book. Store the book name, author, publisher and the year it was published. As with variable names make sure that you key names are clear and descriptive. Print out each piece of information that is stored in your dictionary.
- Add a new book to your dictionary and print it out.

Looping through a dictionary

So far our dictionaries have been small with just one or two sets of key-value pairs. But dictionaries in Python can contain millions of pairs. To handle this amount of information, Python allows you to loop through a dictionary. You can loop through a dictionary's key-value pairs, through its keys or through its values. Let's take a look at each one in turn.

Looping through key-value pairs

Let's create a new dictionary that is designed to store information about books that currently have on our bookshelf.

In [ ]:
```
bookshelf = {
    'book_name' : 'steve jobs',
    'book_author' : 'walter isaacson',
    'published_date' : 2015,
}
```

Here we have a dictionary that currently stores one book. We can access any piece of information about *bookshelf* based on what we have already learned about dictionaries. However, if we wanted to see everything stored in this dictionary we could use a loop. Let's take a look:

In [18]:
```
bookshelf = {
    'book_name' : 'steve jobs',
    'book_author' : 'walter isaacson',
    'published_date' : 2015,
}

for key, value in bookshelf.items():
    print(f"\nKey: {key}")
    print(f"Value: {value}")
```

```
Key: book_name
Value: steve jobs

Key: book_author
Value: walter isaacson

Key: published_date
Value: 2015
```

As before our dictionary is created between lines 1 and 5. At line 7 we write a *for* loop. Here we create names for two variables that will hold the key and value in each key-value pair. We can choose any names for these two variables. For example:

In [19]:
```
bookshelf = {
    'book_name' : 'steve jobs',
    'book_author' : 'walter isaacson',
    'published_date' : 2015,
}

for fred, barney in bookshelf.items():
    print(f"\nKey: {key}")
    print(f"Value: {value}")
```

```
Key: published_date
Value: 2015

Key: published_date
Value: 2015

Key: published_date
Value: 2015
```

Not the two best variable names in the world. It is better to use descriptive, meaningful variable names.

After the variables names in line 1 we have the name of the dictionary followed by the method *items()*, which returns a list of key-value pairs. The *for* loop then assigns each of these pairs to the two variables provided.

At lines 8 and 9 we have two print statements. These print out each key and associated value. Note at line 8 in the *print* statement we use the newline characters to print out a blank line before each key-value pair which gives us a nice clean output.

With what we now know about loops in a dictionary let's revisit our _favorite*food* dictionary:

In [20]:
```
favorite_food = {'tony' : 'chicken wings',
                 'jane' : 'pasta',
                 'tom' : 'burgers',
```

```
                'mary' : 'fish',
                }

for name, food in favorite_food.items():
    print(f"{name.title()}'s favorite food is {food}.")
```

```
Tony's favorite food is chicken wings.
Jane's favorite food is pasta.
Tom's favorite food is burgers.
Mary's favorite food is fish.
```

In this example, because the keys always refer to a person's name and the value is always a food, we use the variable names *name* and *food* in the loop instead of *key* and *value*.

At line 7 we instruct Python to loop through each key-value pair in the dictionary. As it does this, the key from each pair is assigned to the variable called *name*, and the value is assigned to the variable called *food*.

Looping through the keys in a dictionary

We've just learned how to loop through the key-value pairs of a dictionary but there will be times when you only need to work with the keys of a dictionary. To do this Python gives us the *keys()* method. Let's take a look:

In [21]:
```
favorite_food = {'tony' : 'chicken wings',
                'jane' : 'pasta',
                'tom' : 'burgers',
                'mary' : 'fish',
                }

for name in favorite_food.keys():
    print(name.title())
```

```
Tony
Jane
Tom
Mary
```

In this example at line 7 we tell Python to take all of the keys form the dictionary, _favorite*food* and assign them one at a time to the variable *name*.

What happens if we heave out the *keys()* method?

In [22]:
```
favorite_food = {'tony' : 'chicken wings',
                'jane' : 'pasta',
                'tom' : 'burgers',
                'mary' : 'fish',
                }

for name in favorite_food:
    print(name.title())
```

```
Tony
Jane
Tom
Mary
```

We get the same output but why? Looping through the keys the default behavior when looping through a dictionary. You can choose to include or omit the *keys()* method. I use it because I think that it helps to make my more code more readable as well as the intention of my code clearer.

In [24]:
```
favorite_food = {'tony' : 'chicken wings',
                 'jane' : 'pasta',
                 'tom' : 'burgers',
                 'mary' : 'fish',
                 }

friends = ['jane', 'tom']
for name in favorite_food.keys():
    print(f"Hi {name.title()}.")

    if name in friends:
        food = favorite_food[name].title()
        print(f"{name.title()}, your favorite food is {food}.")
```

```
Hi Tony.
Hi Jane.
Jane, your favorite food is Pasta.
Hi Tom.
Tom, your favorite food is Burgers.
Hi Mary.
```

In this code example we have accessed the value associated with a key inside a loop by using the current key.

As before we create a dictionary of favorite foods from lines 1 to 5. Then at line 7 we create a list of friends who we would like to display a message to. Inside the loop we print each person's name. At line 11 we check if the current name is in the list *friends*. If it is, *True*, we determine that person's favorite food using the name of the dictionary and the current value of *name* as the key. Finally, we print a out a message to the identified friends.

What about when a key is not in a dictionary?

In [25]:
```
favorite_food = {'tony' : 'chicken wings',
                 'jane' : 'pasta',
                 'tom' : 'burgers',
                 'mary' : 'fish',
                 }

if 'john' not in favorite_food.keys():
    print("John, what is your favorite food?")
```

```
John, what is your favorite food?
```

As well as looping through a dictionary, the *keys()* method also returns a list of all the keys, and at line 7 it checks if 'john' is in this list.

Looping through keys in a particular order

As of Python 3.7 looping through a dictionary returns the items in the same order that they were inserted. There will be times when you want to loop through the items in a dictionary in a different order. Let's take a look at how we can do this.

In [26]:
```
bookshelf = {
    'adrian goldsworthy' : 'in the name of rome',
    'simon baker' : 'ancient rome',
    'niall ferguson' : 'civilization',
}

for book_author in sorted(bookshelf.keys()):
    print(f"{book_author.title()}, wrote a history book.")
```

```
Adrian Goldsworthy, wrote a history book.
Niall Ferguson, wrote a history book.
Simon Baker, wrote a history book.
```

In this example we have used Python's *sorted()* function to display the keys of the dictionary *bookshelf* in alphabetical order.

At line 7 we've wrapped the *sorted()* function around *bookshelf.keys()* method. This tells Python to list all the keys in the dictionary and sort that list before looping through it.

Looping through all the values in a dictionary

In the previous examples we learned how to output the keys of a dictionary but what if we want to output the values? To achieve this we can use Python's *values()* method which will return a list of values in a dictionary without any keys. Let's see how it works:

In [27]:
```
bookshelf = {
    'adrian goldsworthy' : 'in the name of rome',
    'simon baker' : 'ancient rome',
    'niall ferguson' : 'civilization',
}

print("You have the following books on your bookshelf:")
for book in bookshelf.values():
    print(book.title())
```

```
You have the following books on your bookshelf:
In The Name Of Rome
Ancient Rome
Civilization
```

In this example, the *for* statement at line 8 pulls each value from the dictionary and assigns it to the variable *book*.

This code will pull all the values from a dictionary without checking for repeats. Let's examine an example:

In [28]:
```
bookshelf = {
    'in the name of rome' : 'adrian goldsworthy',
```

```
    'ancient rome' : 'simon baker',
    'civilization' : 'niall ferguson',
    'caesar' : 'adrian goldsworthy',
    'the fall of carthage' : 'adrian goldsworthy',
}

print("You have the following authors on your bookshelf:")
for author in bookshelf.values():
    print(author.title())
```

```
You have the following authors on your bookshelf:
Adrian Goldsworthy
Simon Baker
Niall Ferguson
Adrian Goldsworthy
Adrian Goldsworthy
```

As you can see from the output we have the author Adrian Goldsworthy repeated several times. We don't need to know that we have three books by this author only the author name so show would we remove the repeats? Let's see:

In [29]:
```
bookshelf = {
    'in the name of rome' : 'adrian goldsworthy',
    'ancient rome' : 'simon baker',
    'civilization' : 'niall ferguson',
    'caesar' : 'adrian goldsworthy',
    'the fall of carthage' : 'adrian goldsworthy',
}

print("You have the following authors on your bookshelf:")
for author in set(bookshelf.values()):
    print(author.title())
```

```
You have the following authors on your bookshelf:
Simon Baker
Adrian Goldsworthy
Niall Ferguson
```

At line 10 we have wrapped *set()* around *bookshelf.values()*. Using *set()*, Python identifies the unique items in the list and creates a set from them. A set is a collection of items where each item must be unique.

> Sets can be created directly using curly braces and separating each element within with commas

In [30]:
```
books = {'caesar', 'in the name of rome', 'the fall of carthage'}
books
```

Out[30]:
```
{'caesar', 'in the name of rome', 'the fall of carthage'}
```

> How do you know when you're working with a set and not a dictionary? When you are dealing with sets there will be no key-value pairs

Exercises

-Rewrite the code below to loop through the dictionary that displays its, keys and values.

```
- employee_001 = {'name' : 'tony staunton', 'salary' : 20000
}

- print(employee_001['name'])
- print(employee_001['salary'])
```

- Create a dictionary of books that you have read. Use a loop to print out a sentence about each book
- Use a loop to print only the name of the books in your dictionary
- Use a loop to print out the books authors

Nesting

It is possible and sometimes necessary to store several dictionaries in a list, or a list of items as a value within a dictionary. In Python this is called nesting. You can nest dictionaries inside a list, a list of items inside a dictionary, or nest a dictionary within another dictionary.

A list of dictionaries

When we started learning about dictionaries we wrote a dictionary called _employee*001*. What happens if we have more than one employee? Can how we manage the details of 100 employees. One way is make a list of employees that holds each employee as a dictionary of information about that employee. Let's take a look:

In [31]:
```
employee_001 = {'name' : 'tony staunton', 'salary' : 20000}
employee_002 = {'name' : 'jane smith', 'salary' : 30000}
employee_003 = {'name' : 'mary jones', 'salary' : 40000}

employees = [employee_001, employee_002, employee_003]

for employee in employees:
    print(employee)
```

```
{'name': 'tony staunton', 'salary': 20000}
{'name': 'jane smith', 'salary': 30000}
{'name': 'mary jones', 'salary': 40000}
```

In this example we have first created three dictionaries at lines 1, 2, and 3. At line 5 we store each of these dictionaries in a list called *employees*. We then loop through the list and print out each employee.

In [33]:
```
# Create an empty list of cars
cars_for_sale = []

# Create 50 cars
for car_number in range(50):
    new_car = {'color' : 'black', 'transmission' : 'automatic', 'engine type
    cars_for_sale.append(new_car)

# Display the first 10 cars
for car in cars_for_sale[:20]:
    print(car)
print("...")
```

```
# Show total number of cars that have been created
print(f"Total number of cars: {len(cars_for_sale)}")
```

```
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
...
Total number of cars: 50
```

In this code example we create an empty list at line 2. This list will hold all of the cars that we are about to create. At line 5 we use *range()* which generates a series of numbers that tells Python how many times to repeat the loop. Each time the loop runs, a new car is created at line 6, which is then appended to the list _cars_for*sale* at line 7. At line 10, we use a slice to print the first 10 cars. At line 15 we print the length of the list to show that 50 cars have actually been printed.

Each car that is created has the same characteristics but Python treats them as individual and separate objects. This allows us to modify each car individually if necessary.

Imagine you have a computer game that needs 50, 100 or 1000 cars? This could be one method that you use to easily and quickly create them.

How might we work with this group of cars to change some of cars color from black to navy? Or change the engine type from electric to petrol. Let's take a look:

In [34]:
```
# Create an empty list of cars to manufacture
cars = []

# Create 50 cars
for car_number in range(50):
    new_car = {'color' : 'black', 'transmission' : 'automatic', 'engine type
    cars.append(new_car)

for car in cars[:5]:
    if car['color'] == 'black':
        car['color'] = 'navy'
        car['engine type'] = 'petrol'

# Display the first 10 cars
for car in cars[:10]:
```

```
    print(car)
print("...")
```

```
{'color': 'navy', 'transmission': 'automatic', 'engine type': 'petrol'}
{'color': 'navy', 'transmission': 'automatic', 'engine type': 'petrol'}
{'color': 'navy', 'transmission': 'automatic', 'engine type': 'petrol'}
{'color': 'navy', 'transmission': 'automatic', 'engine type': 'petrol'}
{'color': 'navy', 'transmission': 'automatic', 'engine type': 'petrol'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
{'color': 'black', 'transmission': 'automatic', 'engine type': 'electric'}
...
```

We only want to modify the first 5 cars, so at line 9 we loop through a slice that includes only the first 5. We then write an *if* statement to make sure that we are modifying only black cars. If a car is black, we change it to navy and its engine type to 'petrol'. We then display the first 10 cars to show that only the first 5 have been modified.

In Python it is common to store a number of dictionaries in a list with each dictionary containing lots of different information. For example, books by category, products, or customers.

A list in a dictionary

It can sometimes be useful to put a list inside a dictionary rather than putting a dictionary insides a list which is what our previous examples have been doing. If we were to continue with this method all we can really store in our dictionaries are lists about employee's or books or car. But with a dictionary a list of employees can be just one aspect of our employee management system. Imagine you have an employee directory which lists all your employees. You click on one name to view more details. Let's take a look:

In [35]:
```
# Store information about employees
employees = {
    'name' : 'tony staunton',
    'profile' : ['manager', '4 years of experience', 'developer'],
}

# Output employee information
print(f"Employee {employees['name'].title()}'s company profile is as follows

for detail in employees['profile']:
    print(f"\n- {detail}")
```

```
Employee Tony Staunton's company profile is as follows:

- manager

- 4 years of experience

- developer
```

At line 2 we create a dictionary called *employees* that holds information about a particular employee. One key of this dictionary is *'name'*, and its associated value is *'tony staunton'*. The next key is *'profile'*, which has a list that stores all of the relevant details about this employee.

At line 8 we have a *print* statement that displays the employee name before showing us his profile.

At line 10 we print out the profile of the employee. To access the list of the profile, we use the key *'profile'*, and Python takes the list from the dictionary.

The best time to nest a list inside a dictionary is when you want more than one value to be associated with a single key in a dictionary. Let's take a look at another example:

In [36]:
```
employees = {
    'tony staunton' : ['manager', 'developer', '4 years exerience'],
    'jane smith' : ['senior manager', 'tester', '8 years expereince'],
    'frank jones' : ['graduate', 'business analyst', '1 years expereince']
}

for name, details in employees.items():
    print(f"\n{name.title()}'s company profile:")
    for detail in details:
        print(f"\t- {detail}")
```

```
Tony Staunton's company profile:
        - manager
        - developer
        - 4 years exerience

Jane Smith's company profile:
        - senior manager
        - tester
        - 8 years expereince

Frank Jones's company profile:
        - graduate
        - business analyst
        - 1 years expereince
```

At line 1 we create a new dictionary called *employees*. In this dictionary the names of employees are keys and their details are stored in lists which are the values to each key. At line 7 we begin to loop through the dictionary. We have two variable names, *name* and *details*. We use the variable *details* to hold each value from the dictionary, because we know that each value will be a list.

We have another *for* loop at line 9, which sits inside the main dictionary loop. This loop runs through the details of each employee. We could expand the list of details for each employee to hold as much information as we needed.

A dictionary in a dictionary

We can also nest a dictionary within another dictionary. For example we could have a bookshelf with 500 books, each book has a unique title, we could use the title as the keys in the dictionary. We could then store information about each book by using a dictionary as the value associated with the title. Let's take a look:

In [37]:
```
bookshelf = {
    'geting things done' : {
        'author' : 'david allen',
        'genre' : 'productivity',
```

```
        'read' : 'yes',
    },

    'steve jobs' : {
        'author' : 'walter isaacson',
        'genre' : 'biography',
        'read' : 'yes'
    },
}

for title, book_info in bookshelf.items():
    print(f"\nTitle: {title.title()}")
    author = book_info['author']
    genre = book_info['genre']
    read = book_info['read']

    print(f"\tAuthor: {author.title()}")
    print(f"\tGenre: {genre.title()}")
    print(f"\tRead: {read.title()}")
```

```
Title: Geting Things Done
        Author: David Allen
        Genre: Productivity
        Read: Yes

Title: Steve Jobs
        Author: Walter Isaacson
        Genre: Biography
        Read: Yes
```

In this code example we have store three pieces of information about a book within individual dictionaries, which are themselves inside a dictionary. We access this information by looping through the book titles and the dictionary of information associated with each title.

At line 1 we define a dictionary called *bookshelf*. This dictionary has two keys, *'getting things done'* and *'steve jobs*'. The value associated with each key is a dictionary that includes the book's author, genre, and if it has been read or not.

At line 15 we loop through the *bookshelf* dictionary. Python assigns each key to the variable *title*, and the dictionary associated with each title is assigned to the variable _book*info*. At line 16 we print the book title.

At line 17 we start accessing the inner dictionary. The variable _book*info* which contains book information has three keys: *'author'*, *'genre'*, and *'read'*. We use each key to generate formatted information about each book, which we then print out to the screen.

The structure of each dictionary is identical which is not required by Python but makes them easier to work with.

Exercises

- Create several dictionaries, each one representing an animal. In each dictionary include the kind of animal and if it land or water based and if it is capable of flying. Store these dictionaries in list called animals. Loop through your list and print out the details of each animal.

- Create dictionary of your favorite places. Think of three place names to use as keys in the dictionary, and then store information about your favorite place as the values. Loop through the dictionary and print the keys and values to the screen in a sentence like 'My favorite place is Dublin. It is my favorite place because....'
- Go back through dictionary code examples and pick one for you to copy, modify and run. Perhaps instead of a *bookshelf* dictionary you have stamps, or cars, or Lego.