8. Functions

Defining a Function

In [1]:

```
def greet_user():
    """Display as simple greeting."""
    print("Hello!")

greet_user()
```

```
Hello!
```

- the ***def keyword*** defines the function
- the parantheses hold information about, what kind of information the function needs to do the job
- in this case the name of the fuction is *greet_user()*
- it needs no informations to do its job so the *parantheses* are empty
- the parantheses are required, even if no information is needed
- the definition ends in a colon
- line 2 of the code is *docstring*, which describes what the function does
- docstrings are enclosed in triple quotes!
- this function has jsut one job
  - print("Hello!")
- if you *call* the function it does its job
- to call a function you write the name of the function followed by any necessary information in parantheses
- here no more information is needed

In [1]:

```
def greet_user(username):
    """Display a simple greeting."""
    print("Hello, " + username.title() +  "!")

greet_user('ugur')
```

```
Hello, Ugur!
```

- **passing information to a function**
- when def a function, you can enter in the parantheses def information
- it lets the function accept this information when you call it
- you can pass this information when you call it
- this gives the function the information it needs when you call it to execute the print statement
- you can give any kind of information when you call it and it will be prinbted in the print method

- the defined information is called ***parameter*** (username)
- the information which is passed from the user is called ***argument*** ('ugur')

Tasks

- 8-1. Message: Write a function called display_message() that prints one sentence telling everyone what you are learning about in this chapter. Call the function, and make sure the message displays correctly.
- 8-2. Favorite Book: Write a function called favorite_book() that accepts one parameter, title. The function should print a message, such as One of my favorite books is Alice in Wonderland . Call the function, making sure to include a book title as an argument in the function call.

In [50]:

```
# 8-1
def display_message(task):
    """Display simple message."""
    print("I'm doing some " + task.title() + "!")

display_message('functions')
```

```
I'm doing some Functions!
```

In [51]:

```
# 8-2
def favorite_book(title):
    """This will print your favorite book."""
    print("My favorite book is " + title.title() + "!")

favorite_book('the holy bible')
```

```
My favorite book is The Holy Bible!
```

Passing arguments

In [2]:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet('hamster', 'harry')
```

```
I have a hamster.
My hamster's name is Harry.
```

- **postitional arguments**
- writing arguments in order of the parameters
- you can see it from the definition

In [3]:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet('hamster', 'harry')
describe_pet('dog', 'whillie')
```

```
I have a hamster.
My hamster's name is Harry.

I have a dog.
My dog's name is Whillie.
```

- **multiple function calls**
- you can call a function as many times as needed
- this is a efficient way because once the code is written you just have to call it

In [4]:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet('harry', 'hamster')
```

```
I have a harry.
My harry's name is Hamster.
```

- **order matters in posisitional arguements**

In [5]:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet(animal_type='hamster', pet_name='harry')
```

```
I have a hamster.
My hamster's name is Harry.
```

- **keywoard arguments**
- you give the arguments as a pair (parameter='argument') within, when calling the function
- when calling the function we explicitly tell which parameter each argument should be matched with
- the order of the keyword arguments doesn't matter

In [8]:

```
def describe_pet(pet_name, animal_type='dog'):
    """Display information about a pet."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet(pet_name='willie')
```

```
I have a dog.
My dog's name is Willie.
```

- **default values**
- define a value for each parameter, this will be used, when no other argument is given
- the order of the parameters in the def of the function has to be changed!!!
- the default parameters has to be at the end

In [11]:

```
def describe_pet(pet_name, animal_type='dog'):
    """Display information about a pet."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

# A dog named Willie.
describe_pet('willie')
describe_pet(pet_name='willie')

# A hasmter named Harry.
describe_pet('harry', 'hamster')
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster', pet_name='harry')
```

```
I have a dog.
My dog's name is Willie.

I have a dog.
My dog's name is Willie.

I have a hamster.
My hamster's name is Harry.

I have a hamster.
My hamster's name is Harry.

I have a hamster.
My hamster's name is Harry.
```

- **equivalent function calls**
- each of this function calls would have the same output

In [12]:

```
def describe_pet(pet_name, animal_type='dog'):
    """Display information about a pet."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet()
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-12-1b69b9cb9a9b> in <module>
      4     print("My " + animal_type + "'s name is " + pet_name.tit
le() + ".")
      5
----> 6 describe_pet()

TypeError: describe_pet() missing 1 required positional argument: 'p
et_name'
```

- **avoiding argument errors**
- no argument, information is missing

Tasks

- 8-3. T-Shirt: Write a function called make_shirt() that accepts a size and the text of a message that should be printed on the shirt. The function should print a sentence summarizing the size of the shirt and the message printed on it. Call the function once using positional arguments to make a shirt. Call the function a second time using keyword arguments.
- 8-4. Large Shirts: Modify the make_shirt() function so that shirts are large by default with a message that reads I love Python. Make a large shirt and a medium shirt with the default message, and a shirt of any size with a different message.
- 8-5. Cities: Write a function called describe_city() that accepts the name of a city and its country. The function should print a simple sentence, such as Reykjavik is in Iceland. Give the parameter for the country a default value. Call your function for three different cities, at least one of which is not in the default country.

In [55]:

```
# 8-3
def make_shirt(size='l', text='looser'):
    """This function will print the size and the text on the t-shirt."""
    print("The size of the t-shirt is " + size + " and the text should be " + te
xt + "!")

make_shirt()
make_shirt('m', 'winner')
```

```
The size of the t-shirt is l and the text should be looser!
The size of the t-shirt is m and the text should be winner!
```

In [58]:

```
# 8-4
def make_shirt(size='l', text='i love python'):
    """This function will print the size, which is default l and the text on the
t-shirt."""
    print("The size of the t-shirt is by default " + size + " and the text shoul
d be " + text.title() + "!")

make_shirt()
make_shirt('m')
make_shirt('s', 'i hate python')
```

```
The size of the t-shirt is by default l and the text should be I Lov
e Python!
The size of the t-shirt is by default m and the text should be I Lov
e Python!
The size of the t-shirt is by default s and the text should be I Hat
e Python!
```

In [61]:

```
# 8-5
def describe_city(city, country='germany'):
    """This function will print the city and the country!"""
    print("The city of " + city.title() + " is in the country of " + country.tit
le() + ".")

describe_city('stuttgart')
describe_city('berlin')
describe_city('amsterdam', 'netherlands')
```

```
The city of Stuttgart is in the country of Germany.
The city of Berlin is in the country of Germany.
The city of Amsterdam is in the country of Netherlands.
```

Return Values

In [13]:

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = first_name + ' ' + last_name
    return full_name.title()

musician = get_formatted_name('jimmi', 'hendrix')
print(musician)
```

```
Jimmi Hendrix
```

- the function takes the two parameters / arguments and combines them inti a *full_name*
- the formating is happening with the .title() method
- this will be returned in the calling line
- when you call a function you need a variable *musician* where the return value is stored

In [19]:

```
def get_formatted_name(first_name, last_name, middle_name=''):
    """Return a full name, neatly formatted."""
    if middle_name:
        full_name = first_name + ' ' + middle_name + ' ' + last_name
    else:
        full_name = first_name + ' ' + last_name
    return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)

musician = get_formatted_name('john', 'hooker', 'lee')
print(musician)
```

```
Jimi Hendrix
John Lee Hooker
```

- **making an argument optional**
- the middle_name is optional, so it gets a default value and it will be defined at the end
- a non empty string is always True
- so we check if the the string is true with a conditional test
- the middle name has to be the last argument because it is also the last parameter

In [20]:

```
def build_person(first_name, last_name):
    """Return a dictionary of information about a person."""
    person = {'first': first_name, 'last': last_name}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

```
{'first': 'jimi', 'last': 'hendrix'}
```

- **return a dictionary**
- a function can return any kind of information
- this function takes simple textual information and puts it into a more meaningful data structure that lets you work with the information beyond just printing it
- the strings '*jimi*' and '*hendrix*' are now labeled as a first name and last name
- you can easily extend this function with other information like age, middle name

In [73]:

```
def build_person(first_name, last_name, age=''):
    """Return a dictionary of information about a person."""
    person = {'first': first_name, 'last': last_name}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', age=27)
print(musician)

musician = build_person('jimi', 'hendrix')
print(musician)
```

```
{'first': 'jimi', 'last': 'hendrix', 'age': 27}
{'first': 'jimi', 'last': 'hendrix'}
```

- we add a new optinal parameter *age* to the function definition and assign the parameter as empty
- if the call of the function includes a value for this *age* parameter the value is stored in the dictionary

In [6]:

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = first_name + ' ' + last_name
    return full_name.title()

# This is an infinite loop!
while True:
    print("\nPlease tell me your name:")
    f_name = input("First name: ")
    l_name = input("Last name: ")

    formatted_name = get_formatted_name(f_name, l_name)
    print("\Hello, " + formatted_name + "!")
```

Please tell me your name:

```
---------------------------------------------------------------------
-------
KeyboardInterrupt                         Traceback (most recent cal
l last)
/anaconda3/lib/python3.7/site-packages/ipykernel/kernelbase.py in _i
nput_request(self, prompt, ident, parent, password)
    877             try:
--> 878                 ident, reply = self.session.recv(self.stdin_
socket, 0)
    879             except Exception:

/anaconda3/lib/python3.7/site-packages/jupyter_client/session.py in
recv(self, socket, mode, content, copy)
    802         try:
--> 803             msg_list = socket.recv_multipart(mode, copy=copy
)
    804         except zmq.ZMQError as e:

/anaconda3/lib/python3.7/site-packages/zmq/sugar/socket.py in recv_m
ultipart(self, flags, copy, track)
    469         """
--> 470         parts = [self.recv(flags, copy=copy, track=track)]
    471         # have first part already, only loop while more to r
eceive

zmq/backend/cython/socket.pyx in zmq.backend.cython.socket.Socket.re
cv()

zmq/backend/cython/socket.pyx in zmq.backend.cython.socket.Socket.re
cv()

zmq/backend/cython/socket.pyx in zmq.backend.cython.socket._recv_cop
y()

/anaconda3/lib/python3.7/site-packages/zmq/backend/cython/checkrc.px
d in zmq.backend.cython.checkrc._check_rc()

KeyboardInterrupt:

During handling of the above exception, another exception occurred:

KeyboardInterrupt                         Traceback (most recent cal
l last)
<ipython-input-6-bb56ea58ad3c> in <module>
      7 while True:
      8     print("\nPlease tell me your name:")
----> 9     f_name = input("First name: ")
     10     l_name = input("Last name: ")
     11

/anaconda3/lib/python3.7/site-packages/ipykernel/kernelbase.py in ra
w_input(self, prompt)
    851             self._parent_ident,
    852             self._parent_header,
--> 853             password=False,
    854         )
    855

/anaconda3/lib/python3.7/site-packages/ipykernel/kernelbase.py in _i
nput_request(self, prompt, ident, parent, password)
    881             except KeyboardInterrupt:
```

```
    882                 # re-raise KeyboardInterrupt, to truncate tr
aceback
--> 883                 raise KeyboardInterrupt
    884             else:
    885                 break

KeyboardInterrupt:
```

- here we didn't defined a quit condition
- with a break statement we can do that
- here we have a infitine loop which we want to avoid

In [7]:

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = first_name + ' ' + last_name
    return full_name.title()

# This is an infinite loop!
while True:
    print("\nPlease tell me your name:")
    print("(enter 'q' at any time to quit)")

    f_name = input("First name: ")
    if f_name == 'q':
        break

    l_name = input("Last name: ")
    if l_name == 'q':
        break

    formatted_name = get_formatted_name(f_name, l_name)
    print("\Hello, " + formatted_name + "!")
```

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: Ugur
Last name: q
```

- the **break** statement offers a straightforward way to exit the loop
- we add a message which informs the user how to quit

Tasks

- 8-6. City Names: Write a function called city_country() that takes in the name of a city and its country. The function should return a string formatted like this: "Santiago, Chile" Call your function with at least three city-country pairs, and print the value that's returned.
- 8-7. Album: Write a function called make_album() that builds a dictionary describing a music album. The function should take in an artist name and an album title, and it should return a dictionary containing these two pieces of information. Use the function to make three dictionaries representing different albums. Print each return value to show that the dictionaries are storing the album information correctly. Add an optional parameter to make_album() that allows you to store the number of tracks on an album. If the calling line includes a value for the number of tracks, add that value to the album's dictionary. Make at least one new function call that includes the number of tracks on an album.
- 8-8. User Albums: Start with your program from Exercise 8-7. Write a while loop that allows users to enter an album's artist and title. Once you have that information, call make_album() with the user's input and print the dictionary that's created. Be sure to include a quit value in the while loop.

In [64]:

```
# 8-6
def city_country(city, country):
    """Returns city-country pairs."""
    pair_name = city + ', ' + country
    return pair_name.title()

formatted_pair = city_country

pair = city_country('brussels', 'belgium')
print(pair)

pair = city_country('berlin', 'germany')
print(pair)

pair = city_country('istanbul', 'turkey')
print(pair)
```

```
Brussels, Belgium
Berlin, Germany
Istanbul, Turkey
```

In [76]:

```
# 8-7
def make_album(artist_name, album_title, tracks=''):
    """Return a dictionary of information about an album."""
    information = {'artist': artist_name, 'album title': album_title}
    if tracks:
        information['tracks'] = tracks
    return information

album = make_album('savas', 'beste tag meines lebens')
print(album)

album = make_album('azad', 'leben', 7)
print(album)

album = make_album('jeff mills', 'best of')
print(album)
```

```
{'artist': 'savas', 'album title': 'beste tag meines lebens'}
{'artist': 'azad', 'album title': 'leben', 'tracks': 7}
{'artist': 'jeff mills', 'album title': 'best of'}
```

In [98]:

```
# 8-8
# This is an infinite loop!
def make_album(artist_name, album_title, tracks=''):
    """Return a dictionary of information about an album."""
    information = {'artist': artist_name, 'album title': album_title}
    if tracks:
        information['tracks'] = tracks
    return information

while True:
    print("\nPlease tell me the information:")
    print("(enter 'q' at any time to quit)")

    artist_name = input("Artist name: ")
    if artist_name == 'q':
        break

    album_title = input("Album title: ")
    if album_title == 'q':
        break
    album = make_album(artist_name, album_title)
    print(album)
```

```
Please tell me the information:
(enter 'q' at any time to quit)
Artist name: azad
Album title: leben
{'artist': 'azad', 'album title': 'leben'}

Please tell me the information:
(enter 'q' at any time to quit)
Artist name: q
```

Passing a list

In [8]:

```
def greet_users(names):
    """Print a simple greeting to each user in the list."""
    for name in names:
        msg = "Hello, " + name.title() + "!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

```
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

- we pass a list to a function
- the function gets direct access to the contents of the list
- list of users will be eachwise printed with the greeting
- we degine **greet_users()** so it expects a list of names which it stores in the parameter **names**
- the function loops through the list of users and then pass the list usernames to **greet_users()**
- we are passing the list to the function

In [9]:

```
# Start with some designs that need to be printed.
unprinted_designs = ['iphone case', 'robot pendant', 'dodecahedron']
completed_models = []

# Simulate printing each design, until none are left.
# Move each design to completed_models after printing.
while unprinted_designs:
    current_design = unprinted_designs.pop()

    # Simulate printing each design, until none are left.
    print("Printing model: " + current_design)
    completed_models.append(current_design)

# Display all completed models.
print("\nThe following models have been printed:")
for completed_model in completed_models:
    print(completed_model)
```

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: iphone case

The following models have been printed:
dodecahedron
robot pendant
iphone case
```

- **modifying a list in a function**
- this program starts with a list of designs that need to be printed and an empty list **completed_models**
- after each design is printed it will be moved to the empty list
- with **pop()** and **append()** we move each design from the **unprinted_design** list to the **completed_models** list

In [10]:

```
def print_models(unprinted_designs, completed_models):
    """
    Simulate printing each design, until none are left.
    Move each design to completed_models after printing.
    """
    while unprinted_designs:
        current_design = unprinted_designs.pop()

        # Simulate creating a 3D print from the design.
        print("Printing model: " +  current_design)
        completed_models.append(current_design)

def show_completed_models(completed_models):
    """Show all the models that were printed."""
    print("\nThe following models have been printed:")
    for completed_model in completed_models:
        print(completed_model)
```

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: iphone case

The following models have been printed:
dodecahedron
robot pendant
iphone case
```

- we reorganize the code above with two functions
- each of them has a specific job
- the first function **print_models** works with two lists
- it pop()s from the last item of the list and append()s it to the empty one
- the second function though has just one parameter, the list of completed models
- it prints each name from the completed models list
- this program has the same output like the one above, but the code is **much more organized** because we have 2 seperate functions which makes the main part of the program **easier to understand**

In [11]:

```
unprinted_designs = ['iphone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: iphone case

The following models have been printed:
dodecahedron
robot pendant
iphone case
```

- the main of the program is much easier to read
- we just have two functions
- the first one needs just two arguments (the two lists)
- the second one need just one argument (the completed models list)
- if we need to print the designs we just call the function again
- if we want to modify the function we do it once
- this technique is more efficient then having to update the code seperately in several places in the program
- **every function should have one specific job**

In [17]:

```
unprinted_designs = ['iphone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs[:], completed_models)
show_completed_models(completed_models)
```

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: iphone case

The following models have been printed:
dodecahedron
robot pendant
iphone case
```

- **preventing a Function from Modifying a List**
- passing a function a copy of the list, to prevent the function modifying the original list
- function_name(list_name[:])
  - like this you can prevent
  - the slice notation makes a copy of the list
- the function print_models can work with the **copy of the list_name**

In [101]:

```
def make_pizza(*toppings):
    """Print the list of toppings that have been requested."""
    print(toppings)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

```
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')
```

- 8-9. Magicians: Make a list of magician's names. Pass the list to a function called show_magicians(), which prints the name of each magician in the list.
- 8-10. Great Magicians: Start with a copy of your program from Exercise 8-9. Write a function called make_great() that modifies the list of magicians by adding the phrase the Great to each magician's name. Call show_magicians() to see that the list has actually been modified.
- 8-11. Unchanged Magicians: Start with your work from Exercise 8-10. Call the function make_great() with a copy of the list of magicians' names. Because the original list will be unchanged, return the new list and store it in a separate list. Call show_magicians() with each list to show that you have one list of the original names and one list with the Great added to each magician's name.

In [104]:

```
# 8-9

def show_magicians(magician_names):
    """Print each magicians name in the list."""
    for name in magician_names:
        msg = "Hello, " + name.title() + "!"
        print(msg)

magician_names = ['abra', 'kadabra', 'simsala', 'bim']
show_magicians(magician_names)
```

```
Hello, Abra!
Hello, Kadabra!
Hello, Simsala!
Hello, Bim!
```

In [34]:

```
# 8-10

def show_magicians(magicians):
    """Print the name of each magician in the list."""
    for magician in magicians:
        print(magician)

def make_great(magicians):
    """Add 'the Great!' to each magician's name."""
    # Build a new list to holf the freat magicians.
    great_magicians = []

    # Make each magician great, and add it to great_magicians.
    while magicians:
        magician = magicians.pop()
        great_magician = magician + ' the Great'
        great_magicians.append(great_magician)

    # Add the great magician back into magicians.
    for great_magician in great_magicians:
        magicians.append(great_magician)

magicians = ['Abra', 'Kadabra', 'Simsala', 'Bim']
show_magicians(magicians)

print("\n")
make_great(magicians)
show_magicians(magicians)
```

```
Abra
Kadabra
Simsala
Bim


Bim the Great
Simsala the Great
Kadabra the Great
Abra the Great
```

In [44]:

```
# 8-11

def show_magicians(magicians):
    """Print the name of each magician in the list."""
    for magician in magicians:
        print(magician)

def make_great(magicians):
    """Add 'the Great!' to each magician's name."""
    # Build a new list to hold the great magicians.
    great_magicians = []

    # Make each magician great, and add it to great_magicians.
    while magicians:
        magician = magicians.pop()
        great_magician = magician + ' the Great'
        great_magicians.append(great_magician)

    # Add the great magician back into magicians.
    for great_magician in great_magicians:
        magicians.append(great_magician)

    return magicians

magicians = ['Abra', 'Kadabra', 'Simsala', 'Bim']
show_magicians(magicians)

print("\nGreat magicians:")
great_magicians = make_great(magicians[:])
show_magicians(great_magicians)


print("\nOriginal magicians:")
show_magicians(magicians)
```

```
Abra
Kadabra
Simsala
Bim

Great magicians:
Bim the Great
Simsala the Great
Kadabra the Great
Abra the Great

Original magicians:
Abra
Kadabra
Simsala
Bim
```

Passing an arbitrary number of arguments

- if you won't know how many arguments a function will need you can give the function an arbitrary number of arguments with the asterisk in the parameter
- the arguments will be packed into a tuple
- even if you have just one argument, it will be packed into a tuple

In [20]:

```
def make_pizza(*toppings):
    """Summarize the pizza we are about to make."""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print("- " + topping)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

```
Making a pizza with the following toppings:
- pepperoni

Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

- we replace the print statement with a loop that runs through the list of toppings and describe the pizza which is ordered
- the function responds, whether it receives one value or three

In [21]:

```
def make_pizza(size, *toppings):
    """Summarize the pizza we are about to make."""
    print("\nMaking a " + str(size) +
          "-inch pizza with the following toppings:")
    for topping in toppings:
        print("- " + topping)

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

- **mixing positional and arbitrary arguments**
- positional arguments first, remaining arguments in the final parameter
- in the definition of the function the first value is size
- all values coming after stored in the tuple toppings
- now each piece of information is printed in the proper place showing size first and topping after

In [23]:

```
def build_profile(first, last, **user_info):
    """Build a dictionary containing everything we know
    about a user."""
    profile = {}
    profile['first_name'] = first
    profile['last_name'] = last
    for key, value in user_info.items():
        profile[key] = value
    return profile

user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='pysics')
print(user_profile)
```

```
{'first_name': 'albert', 'last_name': 'einstein', 'location': 'princ
eton', 'field': 'pysics'}
```

- **using arbitrary keyword arguments**
- if you won't know ahead of time what kind of information will be passed to the function
- you know you'll get information about a user, but you are not sure what kind of information you'll receive
- this function always takes a first and last name
- but it accepts an arbitrary number of keyword arguments as well
- the definition of the function expects a first an last name
- **the double asterisks** before the parameter ****user_info** cause to create an empty dictionary calles user_info and pack whatever name-value pairs it receive into a dictionary
- in the body of our function we make an empty dictionary called profile to hold the user's profile
- we add the first and last names to the dictionary because we alleways receive those kind of information
- we loop through the key value pairs in the dictionary **user_info** and add each pair to the profile dictionary
- finally we return the profile dictionary to the function call line
- we call build_profile, pass it the first name and last name and two key-value paors *location= 'princeton* and *field= 'physics'*
- we store the returned profile in user_profile and print user_profile

Tasks

- 8-12. Sandwiches: Write a function that accepts a list of items a person wants on a sandwich. The function should have one parameter that collects as many items as the function call provides, and it should print a summary of the sandwich that is being ordered. Call the function three times, using a different number of arguments each time.
- 8-13. User Profile: Start with a copy of user_profile.py from page 153. Build a profile of yourself by calling build_profile(), using your first and last names and three other key-value pairs that describe you.
- 8-14. Cars: Write a function that stores information about a car in a dictionary . The function should always receive a manufacturer and a model name. It should then accept an arbitrary number of keyword arguments. Call the function with the required information and two other name-value pairs, such as a color or an optional feature. Your function should work for a call like this one:

```
car = make_car('subaru', 'outback', color='blue', tow_pack
age=True)
```

Print the dictionary that's returned to make sure all the information was stored correctly.

In [48]:

```
# 8-12
def make_sandwich(*items):
    """Summarize the sandwich we are about to make."""
    print("\nMaking a sandwich with the following items:")
    for item in items:
        print("- " + item)

make_sandwich('salat')
make_sandwich('cheese', 'green peppers', 'eggs')
make_sandwich('fish')
```

```
Making a sandwich with the following items:
- salat

Making a sandwich with the following items:
- cheese
- green peppers
- eggs

Making a sandwich with the following items:
- fish
```

In [52]:

```
# 8-13
def build_profile(first, last, **user_info):
    """Build a dictionary containing everything we know
    about a user."""
    profile = {}
    profile['first_name'] = first
    profile['last_name'] = last
    for key, value in user_info.items():
        profile[key] = value
    return profile

user_profile = build_profile('ugur', 'tigu',
                             location='aachen',
                             field='data science',
                            job='jobless',
                            sex='male')
print(user_profile)
```

```
{'first_name': 'ugur', 'last_name': 'tigu', 'location': 'aachen', 'f
ield': 'data science', 'job': 'jobless', 'sex': 'male'}
```

- 8-14. Cars: Write a function that stores information about a car in a dictionary . The function should always receive a manufacturer and a model name. It should then accept an arbitrary number of keyword arguments. Call the function with the required information and two other name-value pairs, such as a color or an optional feature. Your function should work for a call like this one:

  ```
  car = make_car('subaru', 'outback', color='blue', tow_pack
  age=True)
  ```

  Print the dictionary that's returned to make sure all the information was stored correctly.

In [53]:

```
def make_car(manufacturer, model_name, **car_info):
    """Build a dictionary containing everything we know about a car."""
    car = {}
    car['manufacturer'] = manufacturer
    car['model name'] = model_name
    for key, value in car_info.items():
        car[key] = value
    return car

car_profile = make_car('audi', 'a4', model_type='sport', production='germany')

print(car_profile)
```

```
{'manufacturer': 'audi', 'model name': 'a4', 'model_type': 'sport',
'production': 'germany'}
```

Storing Your Functions in Modules

In [36]:

```
%%writefile pizza.py
def make_pizza(size, *toppings):
    """Sumarize the pizza we are about to make."""
    print("\nMaking a " + str(size) +
          "-inch pizza with the following toppings:")
    for topping in toppings:
        print("- " + topping)
```

```
Overwriting pizza.py
```

In [38]:

```
import pizza
pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

- **importing an entire module**
- we first write a file with the name ***pizza.py***
- this file is our function
- now we can use it from anywhere just by importing it
- the module is imported
- the line import pizza tells, to open the file and copy all the functions from it into this program
- you don't actually see the code copied
- **to call the function from an imported module, enter the name of the module you imported followed by tha dot**
- this produces the same output as the imported one
    - like this: **module_name.function_name()**

In [41]:

```
from pizza import make_pizza

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

- **importing specific functions**
- you can also import just a specific function from a module
  - from module_name import function_0, function_1
- you can import as many functions you like
- with this syntax, you don't need to use the dot notation, because we declared the function once in the beginning

In [42]:

```
from pizza import make_pizza as mp

mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

- **using 'as' to give a function an alias**
- we change the function make_pizza as mp
- this alias makes it simpler
- the import renames make_pizza to mp and we can use it like mp()
  - from module_name import function_name as fn
  - this is the general syntax

In [43]:

```
import pizza as p

p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

- **using 'as' to give a module an alias**
- you can also do the same for a module
  - import module_name as mn
  - this is the general syntax

In [44]:

```
from pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

- **importing all functions in a module**
- just use the astersk in the import statement
- because every function is imported you can call each function by name, without to use the dot
- **however the best approach is to import the function(s) you want or import the entire module and use the dot**
  - from module_name import *
  - this is the general syntax

Styling Functions

- functions should've descriptive names
- this names should use lowercase letters and underscores
- use comments what the function does
- def function_name(parameter_0, parameter_1='default value')
  - if you use default values no whitespace after and before the equal sign

Tasks

- 8-15. Printing Models: Put the functions for the example print_models.py in a separate file called printing_functions.py. Write an import statement at the top of print_models.py, and modify the file to use the imported functions.
- 8-16. Imports: Using a program you wrote that has one function in it, store that function in a separate file . Import the function into your main program file, and call the function using each of these approaches: import module_name from module_name import function_name from module_name import function_name as fn import module_name as mn from module_name import *
- 8-17. Styling Functions: Choose any three programs you wrote for this chapter, and make sure they follow the styling guidelines described in this section.

In [62]:

```
%%writefile printing_functions.py

# 8-15

def print_models(unprinted_designs, completed_models):
    """
    Simulate printing each design, until none are left.
    Move each design to completed_models after printing.
    """
    while unprinted_designs:
        current_design = unprinted_designs.pop()

        # Simulate creating a 3D print from the design.
        print("Printing model: " +  current_design)
        completed_models.append(current_design)

def show_completed_models(completed_models):
    """Show all the models that were printed."""
    print("\nThe following models have been printed:")
    for completed_model in completed_models:
        print(completed_model)
```

```
Overwriting printing_functions.py
```

In [65]:

```
%%writefile sandwich.py
# 8-16

def make_sandwich(*items):
    """Summarize the sandwich we are about to make."""
    print("\nMaking a sandwich with the following items:")
    for item in items:
        print("- " + item)
```

```
Overwriting sandwich.py
```

In [68]:

```
import sandwich
sandwich.make_sandwich('eggs', 'fish')
```

```
Making a sandwich with the following items:
- eggs
- fish
```

In [69]:

```
from sandwich import make_sandwich
make_sandwich('eggs', 'fish')
```

```
Making a sandwich with the following items:
- eggs
- fish
```

In [70]:

```
import sandwich as s
s.make_sandwich('eggs', 'fish')
```

```
Making a sandwich with the following items:
- eggs
- fish
```