Practical Title: Remote Objects.
Learning outcomes: Learn how to program remote objects using Python.
Software requirements: Python 3
Pyro5 (https://pyro5.readthedocs.io/en/latest/index.html)

To install the Pyro5 library in a pipenv virtual environment use the following commands:
pipenv install pyro5

To run the Python visit.py pipenv virtual environment use the following command:
pipenv run python visit.py

1. Key concepts
In this lab we use a python framework called Pyro5 (Python Remote Objects). The following are a list of key concepts you will encounter when using Pyro5:

Proxy
A proxy is a substitute object for "the real thing" (in the lecture we called this a stub). It intercepts the method calls you would normally do on an object as if it was the actual object. Pyro then performs some magic to transfer the call to the computer that contains the real object, where the actual method call is done, and the results are returned to the caller. This means the calling code doesn't have to know if it's dealing with a normal or a remote object, because the code is identical.

URI
This is what Pyro uses to identify every object. (similar to what a web page URL is to point to the different documents on the web). Its string form is like this: "PYRO:" + object name + "@" + server name + port number. This basically encodes the physical locations of distributed objects in the system.

Pyro object
This is a normal Python object but it is registered with Pyro so that you can access it remotely. Pyro objects are written just as any other object but the fact that Pyro knows something about them makes them special, in the way that you can call methods on them from other programs.

Pyro daemon (server)
This is the part of Pyro that listens for remote method calls, dispatches them to the appropriate actual objects, and returns the results to the caller. All Pyro objects are registered in one or more daemons.

Pyro name server
The name server is a utility that provides a phone book for Pyro applications: you use it to look up a "number" by a "name". The name in Pyro's case is the logical name of a remote object. The number is the exact location where Pyro can contact the object. The use of a name server helps to achieve location transparency

2. Starting a name server
While the use of the Pyro name server is optional, we will use it in this lab. It also shows a few basic Pyro concepts, so let us begin by explaining a little about it. Open a console window and execute the following command to start a name server:

pipenv run python -m Pyro5.nameserver

The name server will start and it prints something like:
Not starting broadcast server for localhost.
NS running on localhost:9090 (127.0.0.1)
URI = PYRO:Pyro.NameServer@localhost:9090

The name server has started and is listening on localhost port 9090. It also printed an URI. Remember that this is what Pyro uses to identify every object.

By default, Pyro uses localhost to run programs on, so you can't by mistake expose your system to the outside world. You'll need to tell Pyro explicitly to use something other than localhost. But it is fine for the lab, so we leave it as it is.

3. Interacting with the name server
There's another command line tool that lets you interact with the name server: "nsc" (name server control tool). You can use it, amongst other things, to see what all known registered objects in the name server are. Let's do that right now. Type:

pipenv run python -m Pyro5.nsc list

and it will print something like this:

--------START LIST
Pyro.NameServer --> PYRO:Pyro.NameServer@localhost:9090
--------END LIST

The only object that is currently registered, is the name server itself! (Yes, the name server is a Pyro object itself.

4. Building a Warehouse
Hint: All code of this part of the practical can be found on LearningCentral.

You'll build a simple warehouse that stores items, and that everyone can visit. Visitors can store items and retrieve other items from the warehouse (if they've been stored there).

In this tutorial you'll first write a normal Python program that more or less implements the complete warehouse system, but in vanilla Python code. After that you'll add Pyro support to it, to make it a distributed warehouse system, where you can visit the central warehouse from many different computers.

4.1 Phase 1: a simple prototype
To start with, write the vanilla Python code for the warehouse and its visitors. This prototype is fully working but everything is running in a single process. It contains no Pyro code at all, but shows what the system is going to look like later on.

The Warehouse object simply stores an array of items which we can query, and allows for a person to take an item or to store an item. Here is the code (warehouse.py):

```
class Warehouse(object):
        def __init__(self):
```

```
            self.contents = ["chair", "bike", "flashlight", "laptop", "couch"]

    def list_contents(self):
            return self.contents

    def take(self, name, item):
            self.contents.remove(item)
            print("{0} took the {1}.".format(name, item))

    def store(self, name, item):
            self.contents.append(item)
            print("{0} stored the {1}.".format(name, item))
```

Then there is a Person that can visit the warehouse. The person has a name and can perform deposit and retrieve actions on a particular warehouse. Here is the code (person.py):

```
class Person(object):
    def __init__(self, name):
            self.name = name

    def visit(self, warehouse):
            print("This is {0}.".format(self.name))
            self.deposit(warehouse)
            self.retrieve(warehouse)
            print("Thank you, come again!")

    def deposit(self, warehouse):
            print("The warehouse contains:", warehouse.list_contents())
            item = input("Type a thing you want to store (or empty): ").strip()
            if item:
                    warehouse.store(self.name, item)

    def retrieve(self, warehouse):
            print("The warehouse contains:", warehouse.list_contents())
            item = input("Type something you want to take (or empty): ").strip()
            if item:
                    warehouse.take(self.name, item)
```

Finally you need a small script that actually runs the code. It creates the warehouse and two visitors, and makes the visitors perform their actions in the warehouse. Here is the code (visit.py):

```
# This is the code that runs this example.
from warehouse import Warehouse
from person import Person

warehouse = Warehouse()
janet = Person("Janet")
henry = Person("Henry")
janet.visit(warehouse)
henry.visit(warehouse)
```

Run this simple program. It will output something like this:

> pipenv run python visit.py

This is Janet.
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'couch']
Type a thing you want to store (or empty): television # typed in
Janet stored the television.
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'couch', 'television']
Type something you want to take (or empty): couch # <-- typed in
Janet took the couch.
Thank you, come again!
This is Henry.
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'television']
Type a thing you want to store (or empty): bricks # <-- typed in
Henry stored the bricks.
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'television', 'bricks']
Type something you want to take (or empty): bike # <-- typed in
Henry took the bike.
Thank you, come again!

4.2 Phase 2: first Pyro version

Now you'll use Pyro to turn the warehouse into a standalone component that people from other computers can visit. You'll need to add a couple of lines to the warehouse.py file so that it will start a Pyro server for the warehouse object. The easiest way to do this is to create the object that you want to make available as Pyro object, and register it with a 'Pyro daemon' (the server that listens for and processes incoming remote method calls):

```
daemon = Daemon()
serve({Warehouse: "example.warehouse"}, daemon=daemon, use_ns=False)
```

Start the warehouse in a new console window, it will print something like this:

$ pipenv run python warehouse.py
Object <class '__main__.Warehouse'>:
*   uri = PYRO:example.warehouse@localhost:34109*
Pyro daemon running.

It will become clear what you need to do with this output in a second. You now need to slightly change the visit.py script that runs the thing. Instead of creating a warehouse directly and letting the persons visit that, it is going to use Pyro to connect to the stand alone warehouse object that you started above. It needs to know the location of the warehouse object before it can connect to it. This is the uri that is printed by the warehouse program above (PYRO:example.warehouse@localhost:34109). You'll need to ask the user to enter that uri string into the program, and use Pyro to create a proxy to the remote object:

uri = input("Enter the uri of the warehouse: ").strip()
warehouse = Proxy(uri)

That is all you need to change. Pyro will transparently forward the calls you make on the warehouse object to the remote object, and return the results to your code. So the code will now look like this (visit.py):

```
# This is the code that visits the warehouse.
from Pyro5.api import Proxy
from person import Person

uri = input("Enter the uri of the warehouse: ").strip()
warehouse = Proxy(uri)
janet = Person("Janet")
henry = Person("Henry")
janet.visit(warehouse)
henry.visit(warehouse)
```

Notice that the code of Warehouse and Person classes didn't change at all.

Run the program. It will output something like this:
> pipenv run python visit.py
Enter the uri of the warehouse: PYRO:example.warehouse@0.0.0.0:34109 # copied
This is Janet.
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'couch']
Type a thing you want to store (or empty): television # typed in
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'couch', 'television']
Type something you want to take (or empty): couch # <-- typed in
Thank you, come again!
This is Henry.
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'television']
Type a thing you want to store (or empty): bricks # <-- typed in
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'television', 'bricks']
Type something you want to take (or empty): bike # <-- typed in
Thank you, come again!

And notice that in the other console window, where the warehouse server is running, the following is printed:
Janet stored the television.
Janet took the couch.
Henry stored the bricks.
Henry took the bike.

4.3 Phase 3: final Pyro version
The code from the previous phase works fine and could be considered to be the final program, but is a bit cumbersome because you need to copy-paste the warehouse URI all the time to be able to use it. You will simplify it a bit in this phase by assigning the object a ***logical name*** and registering this name with the Pyro name server.

Okay, stop the warehouse program from phase 2 if it is still running, and check if the name server that you started previously is still running in its own console window (recall, "python -m Pyro5.nameserver" starts the name server).

In warehouse.py locate the statement "serve({Warehouse: "example.warehouse"}, daemon=daemon, use_ns=False)" and change the "ns=False" argument to "ns=True". This tells Pyro to register the objects in question with the name server.

In visit.py remove the input statement that asks for the warehouse uri, and change the way the warehouse proxy is created. Because you are now using a name server you can ask Pyro to locate the warehouse object automatically using the corresponding logical name:

warehouse = Proxy("PYRONAME:example.warehouse")

So the code should look something like this (visit.py):

```
# This is the code that visits the warehouse.
import sys
import Pyro5.errors
from Pyro5.api import Proxy
from person import Person

sys.excepthook = Pyro5.errors.excepthook

warehouse = Proxy("PYRONAME:example.warehouse")
janet = Person("Janet")
henry = Person("Henry")
janet.visit(warehouse)
henry.visit(warehouse)
```

Start the warehouse program again in a separate console window. It will print something like this:

$ pipenv run python warehouse.py
Object <class '__main__.Warehouse'>:
uri = PYRO:obj_7592ffba34bd407fac53f22c7f5bc2cf@localhost:36619
name = example.warehouse
Pyro daemon running.

As you can see the uri is different this time, it now contains some random id code instead of a name. However it also printed an object name. This is the name that is now used in the name server for your warehouse object. Check this with the 'nsc' tool: "pipenv run python -m Pyro5.nsc list" which will print something like:

--------START LIST
Pyro.NameServer --> PYRO:Pyro.NameServer@localhost:9090
metadata: {'class:Pyro5.nameserver.NameServer'}
example.warehouse --> PYRO:obj_7592ffba34bd407fac53f22c7f5bc2cf@localhost:36619
--------END LIST

This means you can now refer to that warehouse object using the *logical name* example.warehouse and Pyro will locate the correct object for you automatically. This is what you changed in the visit.py code so run that now to see that it indeed works!

Removing an item that is not in the warehouse generates an error. Edit the code to prevent this error from occurring. This may be achieved by checking if an item is in the warehouse before attempting to remove it.

The warehouse currently allows multiple items of the same type or name to be stored. Edit the code to ensure only a single item of a given type can be stored.