A CLOUD GURU

Python 3 Scripting for System Administrators

(EXERCISES)

Control Flow

Exercise: Creating and Displaying Variables

Write a Python script that sets the following variables:

1. first_name - Set to your first name
2. last_name - Set to your last name
3. age - Set to your age as an integer
4. birth_date - Set to your birthdate as a string

Using the variables, print the following to the screen when you run the script:

```
My name is FIRST_NAME LAST_NAME.
I was born on BIRTH_DATE, and I’m AGE years old.
```

One possible solution:

```
#!/usr/bin/env python3

first_name = “Kevin”
last_name = “Bacon”
age = 59
birth_date = “07/08/1958”

print(f”My name is {first_name} {last_name}.”)
print(f”I was born on {birth_date}, and I’m {age} years old.”)
```

Intermediate Scripting

Exercise: Working with If/Else

**Create a script that has a single variable you can set at the top called user.
This user is a dictionary containing the keys:**

- 'admin' - a boolean representing whether the user is an admin user.
- 'active' - a boolean representing whether the user is currently active.
- 'name' - a string that is the user's name.

Example:
user = { 'admin': True, 'active': True, 'name': 'Kevin' }
Depending on the values of user print one of the following to the screen when you run the script.

- Print (ADMIN) followed by the user's name if the user is an admin.
- Print ACTIVE - followed by the user's name if the user is active.
- Print ACTIVE - (ADMIN) followed by the user's name if the user is an admin and active.
- Print the user's name if neither active nor an admin.

Change the values of user and re-run the script multiple times to ensure that it works.

One possible solution:

```
#!/usr/bin/env python3.6

user = { 'admin': True, 'active': True, 'name': 'Kevin' }
prefix = ""

if user['admin'] and user['active']:
    prefix = "ACTIVE - (ADMIN) "
elif user['admin']:
    prefix = "(ADMIN) "
elif user['active']:
    prefix = "ACTIVE - "

print(prefix + user['name'])
```

Exercise: Iterating Over Lists

Building on top of the conditional exercise, write a script that will loop through a list of users where each item is a user dictionary from the previous exercise printing out each user's status on a separate line. Additionally, print the line number at the beginning of each line, starting with line 1. Be sure to include a variety of user configurations in the users list.

User Keys:

- ‘admin’ - a boolean representing whether the user is an admin user.
- ‘active’ - a boolean representing whether the user is currently active.
- ‘name’ - a string that is the user’s name.

Depending on the values of the user, print one of the following to the screen when you run the script.

- Print (ADMIN) followed by the user’s name if the user is an admin.
- Print ACTIVE - followed by the user’s name if the user is active.
- Print ACTIVE - (ADMIN) followed by the user’s name if the user is an admin and active.
- Print the user’s name if neither active nor an admin.

One possible solution:

```
#!/usr/bin/env python3.6
```

```
users = [
    { ‘admin’: True, ‘active’: True, ‘name’: ‘Kevin’ },
    { ‘admin’: True, ‘active’: False, ‘name’: ‘Elisabeth’ },
    { ‘admin’: False, ‘active’: True, ‘name’: ‘Josh’ },
    { ‘admin’: False, ‘active’: False, ‘name’: ‘Kim’ },
]

line = 1

for user in users:
    prefix = f”{line} “

    if user[‘admin’] and user[‘active’]:
        prefix += “ACTIVE - (ADMIN) “
    elif user[‘admin’]:
        prefix += “(ADMIN) “
    elif user[‘active’]:
        prefix += “ACTIVE - “

    print(prefix + user[‘name’])
    line += 1
```

Implementing Features with Test Driven Development

Exercise: Creating and Using Functions

Functions are a great way to organize your code for reuse and clarity.
Write a script that does the following:

- Prompts the user for a message to echo.
- Prompts the user for the number of times to repeat the message. If no response is given, then the count should default to 1.
- Defines a function that takes a message and count then prints the message that many times.

To end the script, call the function with the user-defined values to print to the screen.

Here is one possible solution:

Note: we've called strip() on the count so that we will get a "" if the result is whitespace.

```
#!/usr/bin/env python3.6

message = input("Enter a message: ")
count = input("Number of repeats [1]: ").strip()

if count:
    count = int(count)
else:
    count = 1

def multi_echo(message, count):
    while count > 0:
        print(message)
        count -= 1

multi_echo(message, count)
```

Exercise: Using the 'os' Package and Environment Variables

Environment variables are often used for configuring command line tools and scripts.
Write a script that does the following:

- Prints the first ten digits of PI to the screen.
- Accepts an optional environment variable called DIGITS. If present, the script will print that many digits of PI instead of 10.

Note: You'll want to import pi from the math package.
This task will require some more advanced string formatting. You can read the documentation here, but here's an example of how you could print a float to ten digits:
print("%.*f" % (10, my_float))

Here's one possible solution:

```
#!/usr/bin/env python3.6

from os import getenv
from math import pi

digits = int(getenv("DIGITS") or 10)
print("%.*f" % (digits, pi))
```

Integrating Features and Distributing the Project

Exercise: Creating Files Based on User Input

Write a script that prompts the user for:

- A file_name where it should write the content.
- The content that should go in the file. The script should keep accepting lines of text until the user enters an empty line.

After the user enters an empty line, write all of the lines to the file and end the script.

One possible (robust) solution:

```
#!/usr/bin/env python3.6

def get_file_name(reprompt=False):
    if reprompt:
        print("Please enter a file name.")

    file_name = input("Destination file name: ").strip()
    return file_name or get_file_name(True)

file_name = get_file_name()

print(f"Please enter your content. Entering an empty line will write the
content to {file_name}:\n")

with open(file_name, 'w') as f:
    eof = False
    lines = []

    while not eof:
        line = input()
        if line.strip():
            lines.append(f"{line}\n")
        else:
            eof = True

    f.writelines(lines)
    print(f"Lines written to {file_name}")
```

Final Steps

Exercise: Handling Errors When Files Don't Exist

Write a script that does the following:

- Receives a file_name and line_number as command line parameters.
- Prints the specified line_number from file_name to the screen.
  The user will specify this as you would expect, not using zero as the first line.

Make sure that you handle the following error cases by presenting the user with a useful message:

1. The file doesn't exist.
2. The file doesn't contain the line_number specified (file is too short).

One possible solution:

```
#!/usr/bin/env python3.6

import argparse

parser = argparse.ArgumentParser()
parser.add_argument('file_name', help='the file to read')
parser.add_argument('line_number', type=int, help='the line to print from the
file')

args = parser.parse_args()

try:
    lines = open(args.file_name, 'r').readlines()
    line = lines[args.line_number - 1]
except IndexError:
    print(f"Error: file '{args.file_name}' doesn't have {args.line_number}
lines.")
except IOError as err:
    print(f"Error: {err}")
else:
    print(line)
```

Exercise: Interacting with External Commands

It's not uncommon for a process to run on a server and listen to a port. Unfortunately, you sometimes don't want that process to keep running, but all you know is the port that you want to free up. You're going to write a script to make it easy to get rid of those pesky processes.

Write a script that does the following:

- Takes a port_number as its only argument.
- Calls out to lsof to determine if there is a process listening on that port.
    - If there is a process, kill the process and inform the user.
    - If there is no process, print that there was no process running on that port.

Python's standard library comes with an HTTP server that you can use to start a server listening on a port (5500 in this case) with this line:

```
$ python -m http.server 5500
```
Use a separate terminal window/tab to test our your script to kill that process.

Hints:

- You may need to install lsof. Use this command on CentOS:

```
$ sudo yum install -y lsof
```

- Use this line of lsof to get the port information:

```
lsof -n -i4TCP:PORT_NUMBER
```

That will return multiple lines, and the line you want will contain "LISTEN".

- Use the string split() method to break a string into a list of its words.
- You can either use the kill command outside of Python or the os.kill(pid, 9) function.

One possible solution:

```
#!/usr/bin/env python3.6

import subprocess
import os
from argparse import ArgumentParser

parser = ArgumentParser(description='kill the running process listening on a
given port')
parser.add_argument('port', type=int, help='the port number to search for')

port = parser.parse_args().port

try:
    result = subprocess.run(
            ['lsof', '-n', "-i4TCP:%s" % port],
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE)
except subprocess.CalledProcessError:
    print(f"No process listening on port {port}")

else:
    listening = None
```

```
  for line in result.stdout.splitlines():
    if “LISTEN” in str(line):
      listening = line
      break

 if listening:
    # PID is the second column in the output
    pid = int(listening.split()[1])
    os.kill(pid, 9)
    print(f”Killed process {pid}”)
 else:
    print(f”No process listening on port {port}”)
```

Exercise: Setting Exit Status on Error

You’ve now written a few scripts that handle errors, but when the failures happen the status code returned is still a success (0).
Improve your script to kill processes by exiting with an error status code when there isn’t a process to kill.

Here’s the updated version of the previous example solution:

```
#!/usr/bin/env python3.6

import subprocess
import os
from argparse import ArgumentParser
from sys import exit

parser = ArgumentParser(description=’kill the running process listening on a
given port’)
parser.add_argument(‘port’, type=int, help=’the port number to search for’)

port = parser.parse_args().port

try:
    result = subprocess.run(
            [‘lsof’, ‘-n’, “-i4TCP:%s” % port],
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE)
except subprocess.CalledProcessError:
    print(f”No process listening on port {port}”)
    exit(1)
else:
    listening = None

    for line in result.stdout.splitlines():
        if “LISTEN” in str(line):
            listening = line
                 break
  if listening:
        # PID is the second column in the output
        pid = int(listening.split()[1])
        os.kill(pid, 9)
        print(f"Killed process {pid}")
else:
        print(f"No process listening on port {port}")
        exit(1)
```

Exercise: Installing Third-Party Packages

Utilize pip to install the psycopg2 library (a PostgreSQL database library). Be sure to first create and activate a virtualenv before installing the package.

Here's one way that you could use pip to install the package:

```
$ mkdir venvs
$ python3.6 -m venv venvs/pg
$ source venvs/pg/bin/activate
(pg) $ pip install psycopg2
```

Exercise: Utilizing Third-Party Packages

Make sure that you have the requests package installed.
Now, write a script that does the following:

- Accepts a URL and destination file name from the user calling the script.
- Utilizes requests to make an HTTP request to the given URL.
- Has an optional flag to state whether or not the response should be JSON or HTML (HTML by default).
- Writes the contents of the page out to the destination.

Note: You'll want to use the text attribute to get the HTML.

One possible solution:

```
#!/usr/bin/env python3.6

import sys
import json
import requests
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('url', help='URL to store the contents of')
parser.add_argument('filename', help='the filename to store the content under')
parser.add_argument('--content-type', '-c',
                    default='html',
                    choices=['html', 'json'],
                    help='the content-type of the URL being requested')

args = parser.parse_args()

res = requests.get(args.url)

if res.status_code >= 400:
    print(f"Error code received: {res.status_code}")
    sys.exit(1)

if args.content_type == 'json':
  try:
        content = json.dumps(res.json())
    except ValueError:
        print("Error: Content is not JSON")
        sys.exit(1)
else:
    content = res.text

with open(args.filename, 'w', encoding='UTF-8') as f:
    f.write(content)
    print(f"Content written to '{args.filename}'")
```

Exercise: Creating a Python Project

Over the course of the next few exercises, you'll be creating a Python package to manage users on a server based on an "inventory" JSON file. The first step in this process is going to be setting up the project's directory structure and metadata.

Do the following:

1. Create a project folder called hr (short for "human resources").
2. Set up the directories to put the project's source code and tests.
3. Create the setup.py with metadata and package discovery.
4. Utilize pipenv to create a virtualenv and Pipfile.
5. Add pytest and pytest-mock as development dependencies.
6. Set the project up in source control and make your initial commit.

There's more than one way to set up a project, but here's one way that you could. First, set up the project's folder structure:

```
$ mkdir hr
$ cd hr
$ mkdir -p src/hr tests
$ touch src/hr/__init__.py tests/.keep README.rst
```

With the folders setup, you can then utilize pipenv to add dependency management:

Note: Ensure that which has been installed and is in your $PATH

```
$ pipenv --python python3.6 install --dev pytest pytest-mock
```

Here's a good starting point for a setup.py:

```
setup.py
from setuptools import setup, find_packages

with open('README.rst', encoding='UTF-8') as f:
    readme = f.read()

setup(
    name='hr',
    version='0.1.0',
    description='Commandline user management utility',
    long_description=readme,
    author='Your Name',
    author_email='person@example.com',
    packages=find_packages('src'),
    package_dir={'': 'src'},
    install_requires=[]
)
Lastly, initialize the git repository:
$ git init
$ curl https://raw.githubusercontent.com/github/gitignore/master/Python.
gitignore -o .gitignore
$ git add --all .
$ git commit -m 'Initial commit.'
```

Exercise: Test Drive Building a CLI Parser

The ideal usage of the hr command is this:

```
$ hr path/to/inventory.json
Adding user 'kevin'
Added user 'kevin'
Updating user 'lisa'
Updated user 'lisa'
Removing user 'alex'
Removed user 'alex'
```

The alternative usage of the CLI will be to pass a --export flag like so:

```
$ hr --export path/to/inventory.json
```

This --export flag won't take any arguments. Instead, you'll want to default the value of this field to False and set the value to True if the flag is present. Look at the action documentation to determine how you should go about doing this.

For this exercise, Write a few tests before implementing a CLI parser. Ensure the following:

1. An error is raised if no arguments are passed to the parser.
2. No error is raised if a path is given as an argument.
3. The export value is set to True if the --export flag is given.

Here are some example tests:

```
tests/test_cli.py
import pytest

from hr import cli

@pytest.fixture()
def parser():
    return cli.create_parser()

def test_parser_fails_without_arguments(parser):
    """
    Without a path, the parser should exit with an error.
    """
    with pytest.raises(SystemExit):
        parser.parse_args([])

def test_parser_succeeds_with_a_path(parser):
    """
    With a path, the parser should exit with an error.
    """
    args = parser.parse_args(['/some/path'])
    assert args.path == '/some/path'

def test_parser_export_flag(parser):
    """
    The `export` value should default to False, but set
    to True when passed to the parser.
    """
    args = parser.parse_args(['/some/path'])
    assert args.export == False

    args = parser.parse_args(['--export', '/some/path'])
    assert args.export == True
Here's an example implementation for this cli module:
src/hr/cli.py
import argparse

def create_parser():
    parser = argparse.ArgumentParser()
    parser.add_argument('path', help='the path to the inventory file (JSON)')
    parser.add_argument('--export', action='store_true', help='export current
settings to inventory file')
    return parser
```

Exercise: Implementing User Management

Note: This exercise is large and could take some time to complete, but don›t get discouraged. The tool you're building is going to be running on Linux systems, and it's safe to assume that it's going to run via sudo. With this information, it's safe to say that the tool can utilize usermod, useradd, and userdel to keep users on the server up to date.

Create a module in your package to work with user information. You'll want to be able to do the following:

1. Received a list of user dictionaries and ensure that the system's users match.
2. Have a function that can create a user with the given information if no user exists by that name.
3. Have a function that can update a user based on a user dictionary.
4. Have a function that can remove a user with a given username.
5. The create, update, and remove functions should print that they are creating/updating/removing the user before executing the command.

The user information will come in the form of a dictionary shaped like this:

```
{
  'name': 'kevin',
  'groups': ['wheel', 'dev'],
  'password': '$6$HXdlMJqcV8LZ1DIF$LCXVxmaI/ySqNtLI6b64LszjM0V5AfD.
ABaUcf4j9aJWse2t3Jr2AoB1zZxUfCr8SOG0XiMODVj2ajcQbZ4H4/'
}
The password values will be SHA512 encrypted.
Hint: You can generate an encrypted password in Python that is usable
with usermod -p with this snippet:
import crypt

crypt.crypt('password', crypt.mksalt(crypt.METHOD_SHA512))
```

Tools to Consider:

You'll likely want to interface with the following Unix utilities:

- useradd
- usermod
- userdel

Python modules you'll want to research:

- pwd - Password/User database.
- grp - Group database.

Be careful in testing not to delete your own user or change your password to something that you don't know.

Here's an example of how you could implement this module. These are the tests that I used to drive my implementation:

```
tests/test_users.py
import pytest
import subprocess

from hr import users

# encrypted version of 'password'
password = '$6$HXdlMJqcV8LZ1DIF$LCXVxmaI/ySqNtLI6b64LszjM0V5AfD.
ABaUcf4j9aJWse2t3Jr2AoB1zZxUfCr8SOG0XiMODVj2ajcQbZ4H4/'

user_dict = {
    'name': 'kevin',
    'groups': ['wheel', 'dev'],
    'password': password
}

def test_users_add(mocker):
    """
    Given a user dictionary. `users.add(...)` should
    utilize `useradd` to create a user with the password
    and groups.
    """
    mocker.patch('subprocess.call')
    users.add(user_dict)
    subprocess.call.assert_called_with([
        'useradd',
        '-p',
        password,
        '-G',
        'wheel,dev',
        'kevin',
    ])

def test_users_remove(mocker):
    """
    Given a user dictionary, `users.remove(...)` should
    utilize `userdel` to delete the user.
    """
    mocker.patch('subprocess.call')
    users.remove(user_dict)
    subprocess.call.assert_called_with([
        'userdel',
        '-r',
        'kevin',
    ])

def test_users_update(mocker):
    """
    Given a user dictionary, `users.update(...)` should
    utilize `usermod` to set the groups and password for the
```

```
        user.
        """
        mocker.patch('subprocess.call')
        users.update(user_dict)
        subprocess.call.assert_called_with([
            'usermod',
            '-p',
            password,
            '-G',
            'wheel,dev',
            'kevin',
        ])

    def test_users_sync(mocker):
        """
        Given a list of user dictionaries, `users.sync(...)` should
        create missing users, remove extra non-system users, and update
        existing users. A list of existing usernames can be passed in
        or default users will be used.
        """
        existing_user_names = ['kevin', 'bob']
        users_info = [
            user_dict,
            {
                'name': 'jose',
                'groups': ['wheel'],
                'password': password
            }
        ]
        mocker.patch('subprocess.call')
        users.sync(users_info, existing_user_names)

        subprocess.call.assert_has_calls([
            mocker.call([
                'usermod',
                '-p',
                password,
                '-G',
                'wheel,dev',
                'kevin',
            ]),
            mocker.call([
                'useradd',
                '-p',
                password,
                '-G',
                'wheel',
                'jose',
            ]),
            mocker.call([
                'userdel',
                '-r',
                'bob',
            ]),
    ])
```

Notice: Since there were multiple calls made to subprocess.call within the sync test we used a different assertion method called assert_has_calls which takes a list of mocker.call objects. The mocker.call method wraps the content we would otherwise have put in an assert_called_with assertion.

Here is my implementation of this module (with a few helper functions prefixed with underscores):

```
src/hr/users.py
import pwd
import subprocess
import sys

def add(user_info):
    print(f"Adding user '{user_info['name']}'")
    try:
        subprocess.call([
            'useradd',
            '-p',
            user_info['password'],
            '-G',
            _groups_str(user_info),
            user_info['name'],
        ])
    except:
        print(f"Failed to add user '{user_info['name']}'")
        sys.exit(1)

def remove(user_info):
    print(f"Removing user '{user_info['name']}'")
  try:
        subprocess.call([
            'userdel',
            '-r',
            user_info['name']
        ])
  except:
        print(f"Failed to remove user '{user_info['name']}'")
        sys.exit(1)

def update(user_info):
    print(f"Updating user '{user_info['name']}'")

  try:
        subprocess.call([
            'usermod',
            '-p',
            user_info['password'],
            '-G',
            _groups_str(user_info),
            user_info['name'],
        ])
    except:
            print(f"Failed to update user '{user_info['name']}'")
            sys.exit(1)

def sync(users, existing_user_names=None):
    existing_user_names = (existing_user_names or _user_names())
    user_names = [user['name'] for user in users]
    for user in users:
        if user['name'] not in existing_user_names:
            add(user)
        elif user['name'] in existing_user_names:
            update(user)

    for user_name in existing_user_names:
        if not user_name in user_names:
            remove({ 'name': user_name })

def _groups_str(user_info):
    return ','.join(user_info['groups'] or [])

def _user_names():
    return [user.pw_name for user in pwd.getpwall()
            if user.pw_uid >= 1000 and 'home' in user.pw_dir]
```

I utilized the pwd module to get a list of all of the users on the system and determined which ones weren't system users by looking for UIDs over 999 and ensuring that the user's directory was under home. Additionally, the join method on str was used to combine a list of values into a single string separated by commas. This action is roughly equivalent to:

```
index = 0
group_str = ""
for group in groups:
    if index == 0:
        group_str += group
    else:
        group_str += ",%s" % group
    index+=1
```

To manually test this you'll need to (temporarily) run the following from within your project's directory:

```
sudo pip3.6 install -e .
```

Then you will be able to run the following to be able to use your module in a

REPL without getting permissions errors for calling out to usermod, userdel, and useradd:

```
sudo python3.6
```

```
>>> from hr import users
>>> password = '$6$HXdlMJqcV8LZ1DIF$LCXVxmaI/ySqNtLI6b64LszjM0V5AfD.
ABaUcf4j9aJWse2t3Jr2AoB1zZxUfCr8SOG0XiMODVj2ajcQbZ4H4/'
>>> user_dict = {
...     'name': 'kevin',
...     'groups': ['wheel'],
...     'password': password
... }
>>> users.add(user_dict)
Adding user 'kevin'
>>> user_dict['groups'] = []
>>> users.update(user_dict)
Updating user 'kevin'
>>> users.remove(user_dict)
Removing user 'kevin'
>>>
```

Exercise: JSON Parsing and Exporting

Note: This exercise is large and could take some time to complete, but don›t get discouraged. The last module that you'll implement for this package is one for interacting with the user inventory file. The inventory file is a JSON file that holds user information.

The module needs to:

1. Have a function to read a given inventory file, parse the JSON, and return a list of user dictionaries.
2. Have a function that takes a path, and produces an inventory file based on the current state of the system. An optional parameter could be the specific users to export.

Python modules you'll want to research:

- json - Interact with JSON from Python.
- grp - Group database.
- pwd - Password/user database.
- spwd - Shadow Password database. (Used to get current encrypted password)

Example inventory JSON file:

```
[
  {
    "name": "kevin",
    "groups": ["wheel", "dev"],
    "password": "$6$HXdlMJqcV8LZ1DIF$LCXVxmaI/ySqNtLI6b64LszjM0V5AfD.
ABaUcf4j9aJWse2t3Jr2AoB1zZxUfCr8SOG0XiMODVj2ajcQbZ4H4/"
  },
  {
    "name": "lisa",
    "groups": ["wheel"],
    "password": "$6$HXdlMJqcV8LZ1DIF$LCXVxmaI/ySqNtLI6b64LszjM0V5AfD.
ABaUcf4j9aJWse2t3Jr2AoB1zZxUfCr8SOG0XiMODVj2ajcQbZ4H4/"
  },
  {
    "name": "jim",
    "groups": [],
    "password": "$6$HXdlMJqcV8LZ1DIF$LCXVxmaI/ySqNtLI6b64LszjM0V5AfD.
ABaUcf4j9aJWse2t3Jr2AoB1zZxUfCr8SOG0XiMODVj2ajcQbZ4H4/"
  }
]
```

Hint: If you're writing tests for this code you'll need to heavily rely on mocking to make the interactions with modules like grp, pwd, and spwd consistent.

Here are the tests used to drive the example implementation:

```
tests/test_inventory.py
import tempfile

from hr import inventory

def test_inventory_load():
    """
    `inventory.load` takes a path to a file and parses it as JSON
    """
    inv_file = tempfile.NamedTemporaryFile(delete=False)
    inv_file.write(b"""
    [
      {
        "name": "kevin",
        "groups": ["wheel", "dev"],
        "password": "password_one"
      },
      {
        "name": "lisa",
        "groups": ["wheel"],
        "password": "password_two"
      },
      {
        "name": "jim",
        "groups": [],
        "password": "password_three"
      }
    ]
    """)
    inv_file.close()
    users_list = inventory.load(inv_file.name)
    assert users_list[0] == {
        'name': 'kevin',
        'groups': ['wheel', 'dev'],
        'password': 'password_one'
    }
    assert users_list[1] == {
        'name': 'lisa',
        'groups': ['wheel'],
        'password': 'password_two'
    }
    assert users_list[2] == {
        'name': 'jim',
        'groups': [],
        'password': 'password_three'
    }
```

```
def test_inventory_dump(mocker):
    “””
    `inventory.dump` takes a destination path and optional list of users to export then exports
the existing user information.
    “””
    dest_file = tempfile.NamedTemporaryFile(delete=False)
    dest_file.close()

    # spwd.getspnam can’t be used by non-root user normally.
    # Mock the implemntation so that we can test.
    mocker.patch(‘spwd.getspnam’, return_value=mocker.Mock(sp_pwd=’password’))

    # grp.getgrall will return the values from the test machine.
    # To get consistent results we need to mock this.
    mocker.patch(‘grp.getgrall’, return_value=[
        mocker.Mock(gr_name=’super’, gr_mem=[‘bob’]),
        mocker.Mock(gr_name=’other’, gr_mem=[]),
        mocker.Mock(gr_name=’wheel’, gr_mem=[‘bob’, ‘kevin’]),
    ])

    inventory.dump(dest_file.name, [‘kevin’, ‘bob’])

    with open(dest_file.name) as f:
        assert f.read() == “””[{“name”: “kevin”, “groups”: [“wheel”],
“password”: “password”}, {“name”: “bob”, “groups”: [“super”, “wheel”],
“password”: “password”}]”””
```

Notice that we had to jump through quite a few hoops to get the tests to work consistently for the dump function. The test_inventory_dump required so much mocking that it is debatable as to whether or not it’s worth the effort to test.

Here’s the implementation of the module:

```
src/hr/inventory.py
import grp
import json
import spwd

from .helpers import user_names

def load(path):
    with open(path) as f:
        return json.load(f)

def dump(path, user_names=user_names()):
    users = []
    for user_name in user_names:
        password = spwd.getspnam(user_name).sp_pwd
        groups = _groups_for_user(user_name)
        users.append({
            ‘name’: user_name,
            ‘groups’: groups,
            ‘password’: password
        })
    with open(path, ‘w’) as f:
        json.dump(users, f)

def _groups_for_user(user_name):
    return [g.gr_name for g in grp.getgrall() if user_name in g.gr_mem]
```

The default list of user_names for the dump function used the same code that was used previously in the users module so it was extracted into a new helpers module to be used in both.

```
src/hr/helpers.py
import pwd

def user_names():
    return [user.pw_name for user in pwd.getpwall()
            if user.pw_uid >= 1000 and ‘home’ in user.pw_dir]
```

Here’s the updated users module:

```
src/hr/users.py
import pwd
import subprocess
import sys

from .helpers import user_names

def add(user_info):
    print(“Adding user ‘%s’” % user_info[‘name’])
  try:
        subprocess.call([
            ‘useradd’,
            ‘-p’,
            user_info[‘password’],
            ‘-G’,
            _groups_str(user_info),
            user_info[‘name’],
        ])
  except:
        print(“Failed to add user ‘%s’” % user_info[‘name’])
        sys.exit(1)

def remove(user_info):
    print(“Removing user ‘%s’” % user_info[‘name’])
    try:
        subprocess.call([
            ‘userdel’,
            ‘-r’,
            user_info[‘name’]
        ])
  except:
        print(“Failed to remove user ‘%s’” % user_info[‘name’])
        sys.exit(1)

def update(user_info):
    print(“Updating user ‘%s’” % user_info[‘name’])
  try:
        subprocess.call([
            ‘usermod’,
            ‘-p’,
            user_info[‘password’],
            ‘-G’,
            _groups_str(user_info),
            user_info[‘name’],
        ])
  except:
        print(“Failed to update user ‘%s’” % user_info[‘name’])
        sys.exit(1)

def sync(users, existing_user_names=user_names()):
    user_names = [user[‘name’] for user in users]
  for user in users:
        if user[‘name’] not in existing_user_names:
      add(user)
    elif user[‘name’] in existing_user_names:
            update(user)
  for user_name in existing_user_names:
        if not user_name in user_names:
            remove({ ‘name’: user_name })

def _groups_str(user_info):
    return ‘,’.join(user_info[‘groups’] or [])
```

Manually Test the Module

Load the Python3.6 REPL as root to interact with the new inventory module:

```
$ sudo python3.6
>>> from hr import inventory
>>> inventory.dump(‘./inventory.json’)
>>> exit()
```

Now you can look at the new inventory.json file to see that it dumped the users properly.

```
$ cat inventory.json
[{“name”: “kevin”, “groups”: [“wheel”], “password”:
“$6$HXdlMJqcV8LZ1DIF$LCXVxmaI/ySqNtLI6b64LszjM0V5AfD.
ABaUcf4j9aJWse2t3Jr2AoB1zZxUfCr8SOG0XiMODVj2ajcQbZ4H4/”}]
```

Exercise: Creating the Console Script

Now that you’ve implemented all of the functionality that the hr tools needs, it’s time to wire the pieces together and modify the package metadata to create a console script when installed.

1. Implement main function that ties all of the modules together based on input to the CLI parser.
2. Modify the setup.py so that when installed there is an hr console script.

Here’s an example main function that was added to the cli module:

```
src/hr/cli.py
import argparse

def create_parser():
    parser = argparse.ArgumentParser()
    parser.add_argument(‘path’, help=’the path to the inventory file (JSON)’)
    parser.add_argument(‘--export’, action=’store_true’, help=’export current
settings to inventory file’)
    return parser

def main():
    from hr import inventory, users

    args = create_parser().parse_args()

  if args.export:
        inventory.dump(args.path)
  else:
        users_info = inventory.load(args.path)
        users.sync(users_info)
```

Here are the modifications for the setup.py file necessary to create a console script:

```
setup.py
from setuptools import setup, find_packages

with open(‘README.rst’, encoding=’UTF-8’) as f:
    readme = f.read()

setup(
    name=’hr’,
    version=’0.1.0’,
    description=’Commandline user management utility’,
    long_description=readme,
    author=’Your Name’,
    author_email=’person@example.com’,
    packages=find_packages(‘src’),
    package_dir={‘’: ‘src’},
    install_requires=[],
    entry_points={
        'console_scripts': 'hr=hr.cli:main',
    },
)
```

Since you need sudo to run the script you'll want to install it using sudo pip3.6:

```
$ sudo pip3.6 install -e .
$ sudo hr --help
usage: hr [-h] [--export] path

positional arguments:
  path        the path to the inventory file (JSON)

optional arguments:
  -h, --help  show this help message and exit
  --export    export current settings to inventory file
```

Exercise: Building a Wheel Distribution

Now that you know the tool works, it's time to build it for distribution. Build a wheel for the package and use it to install the hr tool on your system.

Note: This package doesn't support Python 2, so it is not a "universal" package.

Extra metadata file:

```
MANIFEST.in
include README.rst
recursive-include tests *.py
```

Using the pipenv shell, this is the command that you would run to build the wheel:

```
(h4-YsGEiW1S) $ python setup.py bdist_wheel
Lastly, here's how you would install this wheel for the root user to be able to
use (run from project directory):
$ sudo pip3.6 install --upgrade dist/hr-0.1.0-py3-none-any.whl
```