



PRIFYSGOL CYMRU
Y Drindod Dewi Sant
UNIVERSITY OF WALES
Trinity Saint David
SWANSEA - ABERTAWE

Applied Computing

Wales Institute of Science and Art (WISA)

Network Programmability (ACCB5029)

Lab 4: Python Programming

ICONS Used in this document:



Do research and complete the task. Need referencing/ in-text citations. Shall be included as a part of your coursework.



Record your Reflection.

 **Networking**
CISCO Academy

Task 1: (Clean Code)**Objectives**

1. Create tasks using Python Programming
2. Review the programs and execute them with adequate reflections.

Task 2: (Unit Testing)**Objectives**

3. Test a python function using unittest
4. Explore the options in the unittest Framework and submit your outcomes in the Discussion Forum available in Moodle.

Task 3: (Parsing)**Objectives**

5. Parse JSON in Python
6. Reflect your understanding on Clean Coding, Unit Testing and Parsing. Try to
 - explain the theories, concepts, etc. meaningfully to show your understanding.
 - examine and break information down into parts, make inferences, compile, compare information.
 - Rather than describing what, you also need to justify: Why? How? When? Who? Where? At what cost? Etc appropriately and adequately.
 - It is crucial to provide evidence that you have reflected. provide justification/evidence for your arguments and judgements.

Required Resources

Host computer with DEVASC VM.

Instructions

Launch the DEVASC VM

1. Create the following task using Python Programming

Warning: It is up to you either you think the logic or refer to Online resources. However, it is highly recommended to use your logical thinking and work on these tasks. Your reflection must be clear describing your work done. If you are not following the advice, it would be difficult to cope with further tasks.

- 1.1. Write a program which takes a string from the user. Your task is to output the number of numbers and the number of letters. For example:

```
Enter string: Hello@ 123!
Numbers: 3
Letters: 5
```

Hint: a string is technically a list of characters. If myString = "Hello", then myString[0] gives "H", myString[1] gives "e", etc.

Hint: there is an easy way you can test whether a single character is a letter or a number – you may wish to do a little research. Alternatively, you can build a list of every letter (remember case sensitivity), another of every number and compare each letter in the string against each.

Reflection: On your own words, explain the written code.



- 1.2. Take a list of items (separated by commas) from the user. Your task is to split this list and output each item on a new line. Remember the above hint about treating a string like a list of characters. A sample of the input/output might be:

```
Enter string: Orange,Apple,Pear
3 items total.
Orange
Apple
Pear
```

Note: this should work for any number of items.

Reflection: On your own words, explain the written code.



- 1.3. Use Functions, Methods, and Classes

Function: As a quick review, recall that a function is an independently defined block of code that is called by name. In the following example, the function called **functionName** is defined and then called. Notice that it is an independent block of code. It is not encapsulated in any other code.

```
# Define the function
```

```
def functionName:
```

```
...blocks of code...
```

```
# Call the function
```

```
functionName()
```

Method: A method, however, cannot be called by itself. It is dependent on the object in which it is defined. In the following example, the class `className` is declared and three methods are defined. The class is instantiated and then each method of the class is called.

Note: This pseudo-code does not explicitly show the class constructor `__init__` method with the self variable. This special method is reviewed below.

```
# Define the class
```

```
class className
```

```
    # Define a method
```

```
    def method1Name
```

```
        ...blocks of code
```

```
    # Define another method
```

```
    def method2Name
```

```
        ...blocks of code
```

```
    # Define yet another method
```

```
    def method3Name
```

```
        ...blocks of code
```

```
# Instantiate the class
```

```
myClass = className()
```

```
# Call the instantiation and associated methods
```

```
myClass.method1Name()
```

```
myClass.method2Name()
```

```
myClass.method3Name()
```

- 1.3.1. Open a new text file and save it as `myHome.py` in your `~/labs/devnet-src/python` directory.
- 1.3.2. Define the function `myHome` with the argument `street` for street name. When the function is called with a specified street name, it prints a statement that includes the city name.

```
def myHome(street):
```

```
    print("I live in " + street + ".")
```

- 1.3.3. Call the function `myHome` passing it different values for `street`, as shown in the following examples.

```
myHome("CityCentre")
```

```
myHome("Cockett")
```

- 1.3.4. Save and run the `myHome.py` file. You should get the following output.

```
devasc@labvm:~/labs/devnet-src/python$ python3 myHome.py
```

```
I live in CityCentre.
```

```
I live in Cockett.
```

- 1.4. Define a Class

- 1.4.1. Define and then instantiate a class with the `__init__()` method.

With reference to our discussion during the lecture, a Python class is used to create objects that have properties and methods.

All Python classes typically include an explicitly defined `__init__()` function, although you can create a class without defining one. The `__init__()` function is always initiated when a class is instantiated. Instantiating a class creates a copy of the class which inherits all the class variables and methods.

Note: Although it is sometimes called the `__init__()` function, it is dependent on the class. Therefore, it is technically a method.

Open a new text file and save it as `myLocation.py`.

- 1.4.2. Define a class with the name `Location` and press Enter. If you are working in VS Code, then the text editor should automatically indent four spaces.

- 1.4.3. Next, define the `__init__()` function. By convention, the first parameter is called `self`. The `self` parameter is a reference to the current instance of the class itself and is used to access variables that belong to the entire class. The `__init__()` function is then assigned any variables the entire class needs. In the following example, define a `name` and `country` variable. Press Enter twice and then backspace twice to the left margin.

```
def __init__(self, name, country):
```

```
    self.name = name
```

```
    self.country = country
```

- 1.4.4. You can test that this class is now ready to use. Instantiate the class by assigning it a name of your choice. Then specify the values for the required class variables `name` and `country`. The following example uses the `Location` class to instantiate a class called `loc` with a `name` and `country` specified by you. Use your `name` and `country`.

```
loc = Location("Your_Name", "Your_Country")
```

- 1.4.5. To verify that the instantiated loc class now has your assigned name and country, add print statements to your script.

```
print(loc.name)

print(loc.country)
```

- 1.4.6. To verify the loc is indeed a class, add the following print statement that will print the data type for loc.

```
print(type(loc))
```

- 1.4.7. Save and run your script. You should get the following output except with your supplied name and country.

```
devasc@labvm:~/labs/devnet-src/python$ python3 myLocation.py
```

```
Your_Name
```

```
Your_Country
```

```
<class '__main__.Location'>
```

- 1.5. Add a method to the Location class.

Now add a method to the Location class that can be called by a programmer when the class is instantiated. In this simple example, create a method to print the statement, "My name is [name] and I live in [country]."

- 1.5.1. Delete the code the begins with the instantiation of the loc class. Your myLocation.py script should now only include the following code.

```
class Location:

    def __init__(self, name, country):

        self.name = name

        self.country = country
```

- 1.5.2. With your cursor at the end of the line self.country = country, press the Enter key twice and backspace once.

```
self.country = country
```

```
|<--Your cursor should be here.
```

- 1.5.3. Define a new method call myLocation and assigned it the self parameter so that the new method can access the variables defined in the __init__() function. Then, define a print statement to print out the string specified above.

Note: The print statement should be on one line.

```
def myLocation(self):

    print("Hi, my name is " + self.name + " and I live in " + self.country + ".")
```

Press the Enter key twice and backspace twice.

- 1.5.4. Save and run your script to make sure there are no errors. You will not get any output yet.
- 1.6. Instantiate the Location class multiple times and call the myLocation method.

Now that you have a class, you can instantiate it as many times as you like providing different values for the class variables each time.

- 1.6.1. Add the following code to your myLocation.py script to instantiate Location class and call the method. You do not need to add the comments.

```
# First instantiation of the class Location

loc1 = Location("Tomas", "Portugal")

# Call a method from the instantiated class

loc1.myLocation()
```

- 1.6.2. Save and run your script. You should get the following output.

```
devasc@labvm:~/labs/devnet-src/python$ python3 myLocation.py
```

```
Hi, my name is Tomas and I live in Portugal.
```

- 1.6.3. Add two more instantiations and then a fourth one where you specify the name and values for your_loc.

```
loc2 = Location("Ying", "China")

loc3 = Location("Amare", "Kenya")

loc2.myLocation()

loc3.myLocation()

your_loc = Location("Your_Name", "Your_Country")

your_loc.myLocation()
```

- 1.6.4. Save and run your script. You should get the following output.

```
devasc@labvm:~/labs/devnet-src/python$ python3 myLocation.py
```

```
Hi, my name is Tomas and I live in Portugal.
```

```
Hi, my name is Ying and I live in China.
```

```
Hi, my name is Amare and I live in Kenya.
```

```
Hi, my name is Your_Name and I live in Your_Country.
```

1.6.5. Reflection:

What errors did you get while doing the above task?



What is your understanding on Functions, Methods, and Classes?



2. Review the below programs and execute them with adequate reflections.

- 2.1. Understand and troubleshoot this code and record the final working outcome. You must justify your work with step-by-step approach in identifying the logic, troubleshooting the errors, and finally making the program to work as intended.

```
# Given a radius value, print the circumference of a circle.
# Formula for a circumference is c = pi * 2 * radius

class Circle:

    def __init__(self, radius):
        self.radius = radius

    def circumference(self):
        pi = 3.14
        circumferenceValue = pi * self.radius * 2
        return circumferenceValue

    def printCircumference(self):
        myCircumference = self.circumference()
        print ("Circumference of a circle with a radius of " + str(self.radius) + " is " + str(myCircumference))

# First instantiation of the Circle class.
circle1 = Circle(2)
# Call the printCircumference for the instantiated circle1 class.
circle1.printCircumference()

# Two more instantiations and method calls for the Circle class.
circle2 = Circle(5)
circle2.printCircumference()

circle3 = Circle(7)
circle3.printCircumference()
```

- 2.2. Understand and troubleshoot this code below and record the final working outcome. You must justify your work with step-by-step approach in identifying the logic, troubleshooting the errors, and finally making the program to work as intended.

```
class Student:

    def __init__(self, name, id):

        self.name = name

        self.id = id

        self.courses = []

        self.grades = {}

    def register(self, course):

        self.courses.add(course)

    def add_grade(self, course, grade):

        self.grades[course] = grades

    def get_grades(self):

        return self.grade

class Course:

    def __init__(self, name):

        self.name = name

        self.students = []

    def add_student(self, student):

        self.students.append(student)

    def get_students(self):

        return self.student
```

```
def main():

    # create students

    alice = Student("Alice", 1)

    bob = Student("Bob", 2)

    charlie = Student("Charlie", 3)

    # create courses

    math = Course("Math")

    physics = Course("Physics")

    chemistry = Course("Chemistry")

    # register students for courses

    math.add_student(alice)

    math.add_student(bob)

    physics.add_student(bob)

    chemistry.add_student(charlie)

    # add grades

    alice.add_grades(math, 90)

    bob.add_grades(math, 85)

    bob.add_grades(physics, 92)

    charlie.add_grades(chemistry, 87)

    # print out student grades

    for student in [alice, bob, charlie]:

        grades = student.get_grade()

        for course, grade in grades.items():

            print(f"{student.name} received a grade of {grade} in {course.name}")

if __name__ == "__main__":

    main()
```

- 2.3. With your newly formed team (with whom you will be working and completing your group work), discuss and create a new program in python implementing Classes, Methods, and Functions. Write a brief note on what the program is intended to do, (like its deliverables, functions, attributes, methods etc), assign tasks to your group members, and complete the task. (Use Git, GitHub). Share the link with your Lecturer. This task will be addressed as Team_Project_Task_1.

Task 2:

3. In this part, you will **use unittest to test a function** that performs a recursive search of a JSON object. The function returns values tagged with a given key. Programmers often need to perform this kind of operation on JSON objects returned by API calls. This test will use three files as summarized in the following table:

File	Description
recursive_json_search.py	This script will include the json_search() function we want to test.
test_data.py	This is the data the json_search() function is searching.
test_json_search.py	This is the file you will create to test the json_search() function in the recursive_json_search.py script.

- 3.1. Review the test_data.py file.

Open the ~/labs/devnet-src/unittest/test_data.py file and examine its contents. This JSON data is typical of data returned by a call to Cisco's DNA Center API. The sample data is sufficiently complex to be a good test. For example, it has dict and list types interleaved.

```
devasc@labvm:~/labs/devnet-src$ more unittest/test_data.py
key1 = "issueSummary"
key2 = "XY&^$#*@!1234%^&"

data = {
    "id": "AWcvsjx864kVeDHDi2gB",
    "instanceId": "E-NETWORK-EVENT-AWcvsjx864kVeDHDi2gB-1542693469197",
    "category": "Warn",
    "status": "NEW",
    "timestamp": 1542693469197,
    "severity": "P1"
```

- 3.2. Create the json_search() function that you will be testing.
- Our function should expect **a key and a JSON object as input parameters** and return a list of matched key/value pairs.
 - Here is the current version of the function that needs to be tested to see if it is working as intended.

- The purpose of this function is to import the test data first. Then it searches for data that matches the key variables in the test_data.py file.
- If it finds a match, it will append the matched data to a list.
 - The print() function at the end prints the contents for the list for the first variable key1 = "issueSummary".

```
from test_data import *

def json_search(key,input_object):

    ret_val=[]

    if isinstance(input_object, dict): # Iterate dictionary

        for k, v in input_object.items(): # searching key in the dict

            if k == key:

                temp={k:v}

                ret_val.append(temp)

            if isinstance(v, dict): # the value is another dict so repeat

                json_search(key,v)

            elif isinstance(v, list): # it's a list

                for item in v:

                    if not isinstance(item, (str,int)): # if dict or list repeat

                        json_search(key,item)

        else: # Iterate a list because some APIs return JSON object in a list

            for val in input_object:

                if not isinstance(val, (str,int)):

                    json_search(key,val)

    return ret_val

print(json_search("issueSummary",data))
```

- 3.2.1. Open the ~/labs/devnet-src/unittest/recursive_json_search.py file.
- 3.2.2. Copy the code above into the file and save it.

Note: Note that the inline comments should not break to the next line. When pasted into the `recursive_json_search.py`, there should be 21 lines of code inclusive of the open comment `# Fill the Python code in this file.`

3.2.3. Run the code.

You should get no errors and output of `[]` indicating an empty list.

If the `json_search()` function was coded correctly (which it is not), this would tell you that there is no data with the "issueSummary" key reported by JSON data returned by the Cisco DNA Center API.

In other words, there are no issues to report.

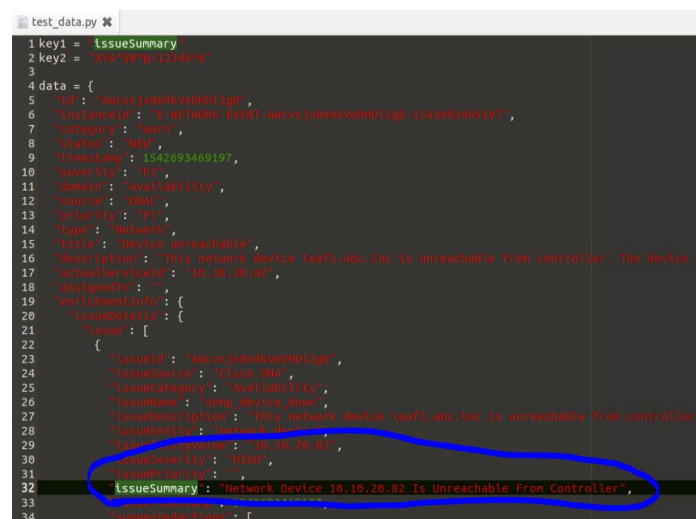
```
devasc@labvm:~/labs/devnet-src/unittest$ python3 recursive_json_search.py
[]
devasc@labvm:~/labs/devnet-src/unittest$
```

3.2.4. But how do you know that the `json_search()` function is working as intended?

You could open the **test_data.py** file and search for the key "issueSummary", as shown below.

If you did, you would indeed find that there is an issue. This is a small data set and a relatively simple recursive search. However, production data and code is rarely this simple.

Therefore, testing code is vital to quickly finding and fixing errors in your code.



```
test_data.py
1 key1 = 'issueSummary'
2 key2 = 'Cisco DNA'
3
4 data = {
5     'id': 'ANUCVJX004V000012g',
6     'instanceId': '1-001000-10001-ANUCVJX004V000012g-1542093469197',
7     'category': 'Norn',
8     'status': 'NOD',
9     'timestamp': '1542093469197',
10    'severity': 'PI',
11    'domain': 'Availability',
12    'source': 'DNA',
13    'priority': 'PI',
14    'type': 'Network',
15    'title': 'Device unreachable',
16    'description': 'This network device leaf2.abc.tnc is unreachable from controller. The device is',
17    'actualServiceId': '10.10.20.82',
18    'assignedTo': '',
19    'enrichmentInfo': {
20        'issueDetails': {
21            'issue': {
22                {
23                    'issueId': 'ANUCVJX004V000012g',
24                    'issueSource': 'Cisco DNA',
25                    'issueCategory': 'Availability',
26                    'issueName': 'Comp. Device down',
27                    'issueDescription': 'This network device leaf2.abc.tnc is unreachable from controller.',
28                    'issueEntity': 'Network Device',
29                    'issueTimestamp': '10.10.20.82',
30                    'issueSeverity': 'High',
31                    'issuePriority': '',
32                    'issueSummary': 'Network Device 10.10.20.82 is Unreachable From Controller',
33                    'issueStatus': 'Unreachable',
34                    'issueDetails': {
```

3.3. Create some unit tests that will test if the function is working as intended.

3.3.1. Open the `~ labs/devnet-src/unittest/test_json_search.py` file.

3.3.2. In the first line of the script after the comment, import the unittest library.

import unittest

3.3.3. Add lines to import the function you are testing as well as the JSON test data the function uses.

```
from recursive_json_search import *
from test_data import *
```

- 3.3.4. Now add the following json_search_test class code to the test_json_search.py file. The code creates the subclass TestCase of the unittest framework. The class defines some test methods to be used on the json_search() function in the recursive_json_search.py script. Notice that each test method begins with test_, enabling the unittest framework to discover them automatically. Add the following lines to the bottom of your ~labs/devnet-src/unittest/test_json_search.py file:

```
class json_search_test(unittest.TestCase):
    """test module to test search function in `recursive_json_search.py`"""
    def test_search_found(self):
        """key should be found, return list should not be empty"""
        self.assertTrue([]!=json_search(key1,data))
    def test_search_not_found(self):
        """key should not be found, should return an empty list"""
        self.assertTrue([]==json_search(key2,data))
    def test_is_a_list(self):
        """Should return a list"""
        self.assertIsInstance(json_search(key1,data),list)
```

In the unittest code, you are using three methods to test the search function:

- ✓ Given an **existing key in the JSON object**, see if the testing code can find such a key.
- ✓ Given a **non-existent key in the JSON object**, see if the testing code confirms that no key can be found.
- ✓ Check if **our function returns a list**, as it should always do.

To create these tests, the script uses some of the built-in assert methods in the unittest TestCase class to check for conditions. The assertTrue(x) method checks if a condition is true and isinstance(a, b) checks if a is an instance of the b type. The type used here is list.

Also, notice that the comments for each method are specified with the triple single quote ("""). This is required if you want the test to output a description of the test method when it runs. Using the single hash symbol (#) for the comment would not print out the description of a failed test.

- 3.3.5. For the last part of the script, add the `unittest.main()` method. This enables running unittest from the command line. The purpose of `if __name__ == '__main__':` is to make sure that the `unittest.main()` method runs only if the script is run directly. If the script is imported into another program, `unittest.main()` will not run. For example, you might use a different test runner than unittest to run this test.

```
if __name__ == '__main__':
    unittest.main()
```

- 3.4. Run the test to see the initial results.
- 3.4.1. Run the test script in its current state to see what results it currently returns. First, you see the empty list. Second, you see the `.F.` highlighted in the output. A period (`.`) means a test passed and an `F` means a test failed. Therefore, the first test passed, the second test failed, and the third test passed.

```
devasc@labvm:~/labs/devnet-src/unittest$ cd /home/devasc/labs/devnet-src/unittest ; /usr/bin/env
hon.python-2023.2.0/pythonFiles/lib/python/debugpy/adapters/../../debugpy/launcher 49761 -- /home/
[]
.F.
=====
FAIL: test_search_found (__main__.json_search_test)
key should be found, return list should not be empty
-----
Traceback (most recent call last):
  File "/home/devasc/labs/devnet-src/unittest/test_json_search.py", line 8, in test_search_found
    self.assertTrue([ ]!=json_search(key1,data))
AssertionError: False is not true
-----
Ran 3 tests in 0.001s

FAILED (failures=1)
```

- 3.4.2. To list each test and its results, run the script again under unittest with the verbose (`-v`) option. Notice that you do not need the `.py` extension for the `test_json_search.py` script. You can see that your test method `test_search_found` is failing.

Note: Python does not necessarily run your tests in order. Tests are run in alphabetical order based on the test method names.

```
devasc@labvm:~/labs/devnet-src/unittest$ python3 -m unittest -v test_json_search
[]
test_is_a_list (test_json_search.json_search_test)
Should return a list ... ok
test_search_found (test_json_search.json_search_test)
key should be found, return list should not be empty ... FAIL
test_search_not_found (test_json_search.json_search_test)
key should not be found, should return an empty list ... ok

=====
FAIL: test_search_found (test_json_search.json_search_test)
key should be found, return list should not be empty
-----
Traceback (most recent call last):
  File "/home/devasc/labs/devnet-src/unittest/test_json_search.py", line 8, in test_search_found
    self.assertTrue([]!=json_search(key1,data))
AssertionError: False is not true
-----

Ran 3 tests in 0.002s

FAILED (failures=1)
```

3.5. Investigate and correct the first error in the recursive_json_search.py script.

3.5.1. The assertion, key should be found, return list should not be empty ... FAIL, indicates the key was not found. Why?

3.5.2. If we look at the text of our recursive function, we see that the statement `ret_val=[]` is being repeatedly executed, each time the function is called. This causes the function to empty the list and lose accumulated results from the `ret_val.append(temp)` statement, which is adding to the list created by `ret_val=[]`.

```
def json_search(key,input_object):
    → ret_val=[]
    if isinstance(input_object, dict): # Iterate dictionary
        for k, v in input_object.items(): # searching key in the dict
            if k == key:
                temp={k:v}
                → ret_val.append(temp)
```

3.5.3. Move the `ret_val=[]` out of our function in recursive_json_search.py so that the iteration does not overwrite the accumulated list each time.

```
from test_data import *
ret_val=[]
def json_search(key,input_object):
```

3.5.4. Save and run the script. You should get the following output which verifies that you resolved the issue. The list is no longer empty after the script runs.

```
devasc@labvm:~/labs/devnet-src/unittest$ cd /home/devasc/labs/devnet-src/unittest
hon.python-2023.2.0/pythonFiles/lib/python/debugpy/adapters/.../debugpy/launcher
h.py
[{'issueSummary': 'Network Device 10.10.20.82 Is Unreachable From Controller'}]
```

3.5.5. Run the test again to see if all errors in the script are now fixed. You got some output last time you ran recursive_json_search.py, you cannot yet be sure you resolved all the errors in the script? Run unittest again without the -v option to see

if `test_json_search` returns any errors. Typically, you do not want to use `-v` option to minimize console output and make tests run faster. At the start of the log you can see `..F`, meaning that the third test failed. Also notice that the list is still printing out. You can stop this behavior by removing the `print()` function in the `recursive_json_search.py` script. But that is not necessary for your purposes in this lab.

```
devasc@labvm:~/labs/devnet-src/unittest$ python3 -m unittest test_json_search
[{'issueSummary': 'Network Device 10.10.20.82 Is Unreachable From Controller'}]
..F
=====
FAIL: test_search_not_found (test_json_search.json_search_test)
key should not be found, should return an empty list
-----
Traceback (most recent call last):
  File "/home/devasc/labs/devnet-src/unittest/test_json_search.py", line 11, in test_search_not_found
    self.assertTrue([]==json_search(key2,data))
AssertionError: False is not true
-----
Ran 3 tests in 0.004s
```

- 3.5.6. Open the `test_data.py` file and search for `issueSummary`, which is the value for `key1`. You should find it twice, but only once in the JSON data object. But if you search for the value of `key2`, which is `XY&^$#@!1234%^&`, you will only find it at the top where it is defined because it is not in the data JSON object. The third test is checking to make sure it is not there. The third test comment states key should not be found, should return an empty list. However, the function returned a non-empty list.
- 3.5.7. Investigate and correct the second error in the `recursive_json_search.py` script. Review the `recursive_json_search.py` code again. Notice that the `ret_val` is now a global variable after you fixed it in the previous step. This means that its value is preserved across multiple invocations of the `json_search()` function. This is a good example of why it's bad practice to use global variables within functions.
- 3.5.8. To resolve this issue, wrap the `json_search()` function with an outer function. Delete your existing `json_search()` function and replace with the refactored one below: (It won't hurt to call the function twice but it's not best practice to repeat a function.)

```
def json_search(key,input_object):
    """
    Search a key from JSON object, get nothing back if key is not found
    key : "keyword" to be searched, case sensitive
    input_object : JSON object to be parsed, test_data.py in this case
    inner_function() is actually doing the recursive search
    return a list of key:value pair
    """
    ret_val=[]
    def inner_function(key,input_object):
        if isinstance(input_object, dict): # Iterate dictionary
            for k, v in input_object.items(): # searching key in the dict
                if k == key:
                    temp={k:v}
                    ret_val.append(temp)
                if isinstance(v, dict): # the value is another dict so repeat
                    inner_function(key,v)
                elif isinstance(v, list):
                    for item in v:
                        if not isinstance(item, (str,int)): # if dict or list repeat
                            inner_function(key,item)
            else: # Iterate a list because some APIs return JSON object in a list
                for val in input_object:
                    if not isinstance(val, (str,int)):
                        inner_function(key,val)
        inner_function(key,input_object)
    return ret_val
print(json_search("issueSummary",data))
```

- 3.5.9. Save the file and run unittest on the directory. You do not need the name of the file. This is because the unittest Test Discovery feature will run any local file it finds whose name begins with test. You should get the following output. Notice that all tests now pass and the list for the "issueSummary" key is populated. You can safely delete the print() function as it would not normally be used when this test is aggregated with other tests for a larger test run.

```
devasc@labvm:~/labs/devnet-src/unittest$ python3 -m unittest
[{'issueSummary': 'Network Device 10.10.20.82 Is Unreachable From Controller'}]
...
-----
Ran 3 tests in 0.001s
OK
```

4. Explore the options in the unittest Framework.
 - 4.1. Submit your outcomes in the Discussion Forum available in Moodle.
 - 4.2. Write a Unit Test for the "Team_Project_Task_1". Make sure, each member of the team is involved in creating a Test case and implement it with necessary reflection and outcome. All your "Team_Project_Task_1" activities must be shared with your Lecturer using GitHub platform.

Task 3: (Parsing)

Parsing means analyzing a message, breaking it into its component parts, and understanding the purpose of each part in context. When messages are transmitted between computers, they travel as a stream of characters. Those characters are effectively a string. That message needs to be parsed into a semantically-equivalent data-structure containing data of recognized types (e.g., integers, floats, strings, and Booleans) before the data can be interpreted and acted upon. In this lab, you will use Python to parse each data format in JSON.

Independent Learning Activity: Try using Python to parse each data format in XML, and YAML.

Parsing **JavaScript Object Notation (JSON)** is a frequent requirement of interacting with REST APIs. The steps are usually as follows:

- Authenticate using a user/password combination to retrieve a token that will expire after a set amount of time. This token is used for authenticating subsequent requests.
- Execute a GET request to the REST API, authenticating as required, to retrieve the state of a resource, requesting JSON as the output format.
- Modify the returned JSON, as needed.
- Execute a POST (or PUT) to the same REST API (again, authenticating as required) to change the state of the resource, again requesting JSON as the output format and interpreting it as needed to determine whether the operation was successful.

In Python scripts, the Python json library can be used to parse JSON into Python native data structures and serialize data structures back out as JSON. The Python yaml library can be used to convert the data to YAML.

The following program uses both modules to parse the above JSON data, extract and print data values, and output a YAML version of the file.

It uses the json library loads() method to parse a string into which the file has been read.

It then uses normal Python data references to extract values from the resulting Python data structure.

Finally, it uses the yaml library dump() function to serialize the Python data back out as YAML, to the terminal.

5. Open the **parsejson.py** file found in the **~/labs/devnet-src/parsing** directory.

```
import json
import yaml
```

- 5.1. Use the Python **with** statement to open **myfile.json** and set it to the variable name **json_file**. Then use the **json.load** method to load the JSON file into a string set to the variable name **ourjson**.

```
# There is no need to explicitly close the file as the with statement ensures proper opening and closing of the file.
with open('myfile.json','r') as json_file:
    ourjson = json.load(json_file)
```

- 5.1.1. Add a print statement for **ourjson** to see that it is now a Python dictionary.

```
print(ourjson)
```

- 5.1.2. Save and run your script. You should see the following output.

```
devasc@labvm:~/labs/devnet-src/unittest$ cd /home/devasc/labs/devnet-src/parsing ; /usr/bin/env /bin/python3 /home/devasc/.vscode/extensions/ms-python.python-2023.2.0/pythonFiles/lib/python/debugpy/adapters/..../debugpy/launcher 57285 -- /home/devasc/labs/devnet-src/parsing/parsejson.py
{'access_token': 'ZDI3MGEyYzQtNmFLNS00NDNhLWFLNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3', 'expires_in': 1209600, 'refresh_token': 'MDEyMzQ1Njc4OTAxMjMONTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4OTEyMzQ1Njc4', 'refresh_token_expires_in': 7776000}
```

- 5.2. Add print statements that display the token value and how many seconds until the token expires.

```
print("The access token is: {}".format(ourjson['access_token']))
print("The token expires in {} seconds.".format(ourjson['expires_in']))
```

- 5.2.1. Save and run your script. You should see the following output.

```
devasc@labvm:~/labs/devnet-src/parsing$ cd /home/devasc/labs/devnet-src/parsing ; /usr/bin/env /bin/python3 /home/devasc/.vscode/extensions/ms-python.python-2023.2.0/pythonFiles/lib/python/debugpy/adapters/..../debugpy/launcher 56079 -- /home/devasc/labs/devnet-src/parsing/parsejson.py
{'access_token': 'ZDI3MGEyYzQtNmFLNS00NDNhLWFLNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3', 'expires_in': 1209600, 'refresh_token': 'MDEyMzQ1Njc4OTAxMjMONTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4', 'refresh_token_expires_in': 7776000}
The access token is: ZDI3MGEyYzQtNmFLNS00NDNhLWFLNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3
The token expires in 1209600 seconds.
```

- 5.3. Add a print statement that will display the three dashes required for a YAML file. The two `\n` will add two lines after the previous output. Then add a statement to print `ourjson` as YAML data by using the `dump()` method of the `yaml` library.

```
print("\n\n---")
print(yaml.dump(ourjson))
```

- 5.3.1. Save and run your script. You should see the following output.

```
sing/parsejson.py
{'access_token': 'ZDI3MGEyYzQtNmFLNS00NDNhLWFLNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3', 'expires_in': 1209600, 'refresh_token': 'MDEyMzQ1Njc4OTAxMjMONTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4', 'refresh_token_expires_in': 7776000}
The access token is: ZDI3MGEyYzQtNmFLNS00NDNhLWFLNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3
The token expires in 1209600 seconds.

---
access_token: ZDI3MGEyYzQtNmFLNS00NDNhLWFLNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3
expires_in: 1209600
refresh_token: MDEyMzQ1Njc4OTAxMjMONTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4OTEyMzQ1Njc4
refresh_token_expires_in: 7776000
```

6. Reflect your understanding on Clean Coding, Unit Testing and Parsing. Try to



- 6.1. explain the theories, concepts, etc. meaningfully to show your understanding.
- 6.2. examine and break information down into parts, make inferences, compile, compare information.
- 6.3. Rather than describing what, you also need to justify: Why? How? When? Who? Where? At what cost? Etc appropriately and adequately.
- 6.4. It is crucial to provide evidence that you have reflected. provide justification/evidence for your arguments and judgements.

Deliverables of this Task (In Moodle, GitHub (shared with your Lecturer), Discussion Forum etc.)

- 1.1. Reflection: On your own words, explain the written code
- 1.2. Reflection: On your own words, explain the written code

1.6.5. Reflection:

What errors did you get while doing the above task?

What is your understanding on Functions, Methods, and Classes?

- 2.1. Understand and troubleshoot this code and record the final working outcome. You must justify your work with step-by-step approach in identifying the logic, troubleshooting the errors, and finally making the program to work as intended.
- 2.2. Understand and troubleshoot this code below and record the final working outcome. You must justify your work with step-by-step approach in identifying the logic, troubleshooting the errors, and finally making the program to work as intended.
- 2.3. With your newly formed team (with whom you will be working and completing your group work), discuss and create a new program in python implementing Classes, Methods, and Functions. Write a brief note on what the program is intended to do, (like its deliverables, functions, attributes, methods etc), assign tasks to your group members, and complete the task. (Use Git, GitHub). Share the link with your Lecturer.
- 4 Explore the options in the unittest Framework.
 - 4.1. Submit your outcomes in the Discussion Forum available in Moodle.
 - 4.2. Write a Unit Test for the "Team_Project_Task_1". Make sure, each member of the team is involved in creating a Test case and implement it with necessary reflection and outcome. All your "Team_Project_Task_1" activities must be shared with your Lecturer using GitHub platform.
6. Reflect your understanding on Clean Coding, Unit Testing and Parsing. Try to
 - 6.1. explain the theories, concepts, etc. meaningfully to show your understanding.
 - 6.2. examine and break information down into parts, make inferences, compile, compare information.
 - 6.3. Rather than describing what, you also need to justify: Why? How? When? Who? Where? At what cost? Etc appropriately and adequately.
 - 6.4. It is crucial to provide evidence that you have reflected. provide justification/evidence for your arguments and judgements.

End of document.