BÀI TẬP VỀ NHÀ

HANDLE EXCEPTION & MODULE/PACKAGE

Bài 6.1:
Xử lý exception sau:

```
# Setup
actor = {"name": "John Cleese", "rank": "awesome"}

# Function to modify!!!
def get_last_name():
    return actor["last_name"]

# Test code
get_last_name()
print("All exceptions caught! Good job!")
print("The actor's last name is %s" % get_last_name())
```

Bài 6.2:
Xử lý exception sau:

```
def do_stuff_with_number(n):
    print(n)

def catch_this():
    the_list = (1, 2, 3, 4, 5)

    for i in range(20):
        try:
            do_stuff_with_number(the_list[i])
        except IndexError: # Raised when accessing a non-existing index of a list
            do_stuff_with_number(0)
```

```
catch_this()
```

Bài 6.3:

Xử lý exception sau:

```
print(a)

#error

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

NameError: name 'a' is not defined

</module></stdin>
```

Bài 6.4:

Write a Python program to generate a random color hex, a random alphabetical string, random value between two integers (inclusive) and a random multiple of 7 between 0 and 70. Use random.randint()

Bài 6.5:

Write a Python program to select a random element from a list, set, dictionary (value) and a file from a directory. Use random.choice()

Bài 6.6:

Write a Python program to count the number of lines in a given CSV file. Use csv.reader

Bài 6.7*:

1. *Basics, import*. Write a program that counts lines and characters in a file (similar in spirit to “wc” on Unix). With your favorite text editor, code a Python module called mymod.py, which exports three top-level names:

   a) A countLines(name) function that reads an input file and counts the number of lines in it (hint: file.readlines() does most of the work for you, and len does the rest)

b) A countChars(name) function that reads an input file and counts the number of characters in it (hint: file.read() returns a single string)

c) A test(name) function that calls both counting functions with a given input filename. Such a filename generally might be passed-in, hard-coded, input with raw_input, or pulled from a command-line via the sys.argv list; for now, assume it's a passed-in function argument.

All three mymod functions should expect a filename string to be passed in. If you type more than two or three lines per function, you're working much too hard—use the hints listed above!

Now, test your module interactively, using import and name qualification to fetch your exports. Does your PYTHONPATH need to include the directory where you created mymod.py? Try running your module on itself: e.g., test("mymod.py"). Note that test opens the file twice; if you're feeling ambitious, you may be able to improve this by passing an open file object into the two count functions (hint: file.seek(0) is a file rewind).

2. *from/from**. Test your mymod module from Exercise 1 interactively, by using from to load the exports directly, first by name, then using the from* variant to fetch everything.

3. __main__. Now, add a line in your mymod module that calls the test function automatically only when the module is run as a script, not when it is imported The line you add will probably test the value of __name__ for the string "__main__", as shown in this unit. Try running your module from the system command line; then, import the module and test its functions interactively. Does it still work in both modes?

4. *Nested imports*. Write a second module, myclient.py, which imports mymod and tests its functions; run myclient from the system command line. If myclient uses from to fetch from mymod, will mymod's functions be accessible from the top level of myclient? What if it imports with import instead? Try coding both variations in myclient and test interactively, by importing myclient and inspecting its __dict__.

5. *Package imports*. Finally, import your file from a package. Create a subdirectory called mypkg nested in a directory on your module import search path, move the mymod.py module file you created in exercises 1 or 3 into the new directory, and try to import it with a package import of the form: import mypkg.mymod.

   You'll need to add an __init__.py file in the directory your module was moved to in order to make this go, but it should work on all major Python platforms (that's part of the reason Python uses "." as a path separator). The package directory you create can be simply a subdirectory of the one you're working in; if it is, it will be found via the home directory component of the search path, and you won’t have to configure your path. Add some code to your __init__.py, and see if it runs on each import.

6. *Reload*. Experiment with module reloads: perform the tests in the changer.py example, changing the called function’s message and/or behavior repeatedly, without stopping the Python interpreter. Depending on your system, you might be able to edit changer in another window, or suspend the Python interpreter and edit in the same window (on Unix, a Ctrl-Z key combination usually suspends the current process, and a fg command later resumes it).

7. *[Optional] Circular imports* (and other acts of cruelty). In the section on recursive import gotchas, importing recur1 raised an error. But if we restart Python and import recur2 interactively, the error doesn’t occur: test and see this for yourself. Why do you think it works to import recur2, but not recur1? (Hint: Python stores new modules in the built-in sys.modules table (a dictionary) before running their code; later imports fetch the module from this table first, whether the module is “complete” yet or not.) Now try running recur1 as a top-level script file: % python recur1.py. Do you get the same error that occurs when recur1 is imported interactively? Why? (Hint: when modules are run as programs they aren’t imported, so this case has the same effect as importing recur2 interactively; recur2 is the first module imported.) What happens when you run recur2 as a script?