CSP1150/CSP5110: Programming Principles
Workshop 4: Functions and Modules

Task 1 – Concept Revision in the Python Shell

Let's revise the concepts from this module by typing some simple statements into the **Python Shell**. Feel free to deviate from the workshop and experiment!

1. `random.randint(1, 20)`
   Try typing this before importing the random module – you should receive an error message.

2. `from random import randint`
3. `random.randint(1, 20)`
4. `randint(1, 20)`
   When you use "from" to import a single thing from a module (in this case, the `randint()` function from the random module), it becomes a local function – you can't refer to it as `random.randint()` since you haven't imported the random module, just one part of it.

5. `import random`
6. `random.randint(1, 20)`
7. `names = ['Sam', 'Sue', 'John', 'Mary', 'Toby']`
8. `random.choice(names)`
9. `random.sample(names, 3)`
   Once you've imported the entire module, you can use any of its functions.

10. `import string`
11. `string.ascii_uppercase`
12. `string.digits`
13. `string.punctuation`
    Import the string module and look into the constants it provides.

14. `lettersAndNumbers = string.ascii_letters + string.digits`
    They are just strings, so you can combine them using concatenation. "string.ascii_letters" itself is just a concatenation of "string.ascii_uppercase" and "string.ascii_lowercase".

15. `random.sample(lettersAndNumbers, 10)`
    This will randomly select 10 different characters from `lettersAndNumbers`.

Task 2 – Enhancing the "convert" Module

Create a new file in the **Python Editor** and copy-paste the following code into it:

```python
MM_IN_INCH = 25.4
MM_IN_FOOT = 304.8
MM_IN_YARD = 914.4
MM_IN_MILE = 1609344

def cm2inches(cm):
    return cm * 0.393

def m2feet(m):
    return m * 3.281

def m2yards(m):
    return m * 1.094

def km2miles(km):
    return km * 0.621
```

(This code is the same as the code in the "Creating a Module" section of Lecture 4)

Save the file as "**convert.py**", then create a **new file** and copy-paste the following code into it:

```python
import convert

print('20 metres is', convert.m2feet(20), 'feet.')

print('4 miles is', convert.MM_IN_MILE * 4, 'millimetres.')
```

(This code is the same as the code in the "Creating a Module" section of Lecture 4)

Save this file (filename not important) in the **same folder** as convert.py, then run the code to test it. Once you've made sure that it is working, **enhance convert.py** to make it more useful in these ways:

1. Add a second parameter named "rounding" to each of the functions. The parameter should have a default value of `2`, and should be used to round the result of the function.
   Test your work by calling the functions in your second file:
   - `convert.cm2inches(7)` should return a value of `2.75`
   - `convert.m2yards(10, 1)` should return a value of `10.9`

2. Add a third parameter named "addUnit" to each of the functions. The parameter should have a default of False. If True is specified for the parameter when the function is called, the function should return a string of the result with the unit of measurement at the end, e.g. `'2.75 inches'` instead of `2.75`. Test your work by calling the functions:
   - `convert.m2feet(5, 0, True)` should return `16.0 feet`
   - `convert.km2miles(2, addUnit=True)` should return `1.24 miles`

Task 3 – Input Type Validating Functions

Create a new file in the **Python Editor** and write the code to implement the following functions.

So far in the unit, we've assumed that the user will input the correct type of data when prompted. Hence, we've done things like converting input to an integer without making sure it actually is one:

```python
intVal = int(input('Enter an integer: '))
```

This works if the user types an integer, but if they don't the program will end with an error:

```
ValueError: invalid literal for int() with base 10: 'd'
```

The "Continue Example 1" from Lecture 3 touched upon a method of ensuring that the user enters a number when prompted, and re-prompts them if they enter something other than a number.

Create a function named "`inputInt()`" which works similar to the built-in "`input()`" function, except that it will repeatedly re-prompt the user for input until they enter an integer. The pseudocode for the function is below. Use the example from Lecture 3 to help you write the code.

```
Define inputInt(prompt) function:
    Endless Loop:
        Prompt for input with prompt parameter

        Try:
            Convert response to int and store in numResponse variable

        On ValueError exception:
            Print 'invalid input' error message
            Continue to next iteration of loop

        Return numResponse variable (ends function / breaks loop)
```

As you can see, if there is an error trying to convert the input to an integer, the error message will be printed and the continue statement will go back to the start of the loop – re-prompting for a value without returning anything. Only when the conversion is successful will the function return/end.

Once you've written the code for the function, *test it*. If you enter anything other than an integer, you should receive an error message and be re-prompted. Once you enter an integer, the program should continue. Use the following code to test your function:

```python
value = inputInt('Enter an int: ')
print('Value is', value)
```

```
Enter an int: no
Invalid input: int required.
Enter an int: 7
Value is 7
```

Once your function is working correctly, copy and paste the code to create a second function in the same file. Name the second function "`inputFloat()`", and make it work the same as "`inputInt()`", but require the user to enter a floating point number.

Task 4 – Enhancing the Input Type Validating Functions

Once both "`inputInt()`" and "`inputFloat()`" are working, enhance both functions in the following ways to make them more useful:

1. Add second parameter named "`errorMessage`" to the functions. The parameter should have a default value of the "invalid input" error message currently used by the functions. This parameter allows a custom error message to be shown instead of the default one.
   Hint: This shouldn't require much new code – just adding a parameter and moving the default error message.
   - `inputInt('Enter an int: ')` would use the default error message
   - `inputInt('Enter your age: ', 'Enter age as an integer.')` would use "Enter age as an integer." as the error message

2. Add two more parameters named "`minValue`" and "`maxValue`" to the functions. Give them default values of `None`, e.g. "`minValue = None`" in the function parameter area. These parameters allow a minimum and maximum allowable value to be specified for the input. If the input is less than the minimum or greater than the maximum (assuming a value has been specified for that parameter), show an appropriate error message.
   Hint: Use a boolean expression of "`minValue is not None and numResponse < minValue`" in an "`if`" statement to check the minimum value (and a similar expression to check the maximum value).

With these two enhancements, the code of the functions should resemble this pseudocode:

```
Define "inputInt" function:                                    Pseudocode
Parameters: prompt (required),
            errorMessage (default of default error message),
            minValue (default of None),
            maxValue (default of None)

    Endless Loop:
        Prompt for input with prompt parameter

        Try:
            Convert response to int and store in numResponse variable

        On ValueError exception:
            Print errorMessage parameter
            Continue to next iteration of loop

        If minValue is not None and numResponse < minValue:
            Print "value below minimum" error message
            Continue to next iteration of loop

        If maxValue is not None and numResponse > maxValue:
            Print "value above maximum" error message
            Continue to next iteration of loop

        Return numResponse variable (ends function / breaks loop)
```

(Note: These functions worked on further in Reading 4.3!)

Task 5 – Improving Prior Code

Once you have completed Task 4 (and ideally Reading 4.3 as well), use the "`inputInt()`" and "`inputFloat()`" functions to improve code you wrote in prior workshops. A number of previous workshop tasks required integers or floats to be entered, but simply assumed that the value received was appropriate and tried to convert it without any error checking.

Treat your file containing the "`inputInt()`" and "`inputFloat()`" functions as a module (remember to remove any testing code from it): Import it into your previous workshop files and replace "`input()`" statements that are converted to int or float with your new functions.

Remember, programming is a cyclical process – it is very rare that you will write a piece of code perfectly the first time. You will often find yourself returning to previously written code once you find a way to refine or enhance it.

That's all for this workshop. If you haven't completed the workshop or readings, find time to do so before next week's class.