ICS3U: Advanced Input Validation

We have previously seen ways in which we can screen user input to ensure that any values entered are acceptable for use, provided that the user entered the correct type of data. For example, attempting to take the square root of a negative number is not possible (at least in the real number system) and will cause our program to crash.

```
>>> import math
>>> math.sqrt(-9)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: math domain error
```

By using a loop, we can redirect the user to enter a positive value instead.

```
n = int(input("Enter a positive integer: "))
while n < 0:
   n = int(input("Integer must be positive, try again: "))
```

If the user *doesn't* enter the correct data type, however, then this code will not work. The problem stems from our typecasting call to `int`. When the Python interpreter attempts to convert the string returned from the `input` function into an integer, it may come across one or more characters that cannot be represented as digits, such as letters, special characters, and so on. Without converting to an integer, however, means that we cannot use the `sqrt` function in the `math` module, since it requires a numeric data type on which to operate. Entering non-numeric data will cause a run-time error.

```
Enter a positive integer: F
Traceback (most recent call last):
  File "/home/Code/input_validation.py", line 1, in <module>
    n = int(input("Enter a positive integer: "))
ValueError: invalid literal for int() with base 10: 'F'
```

One solution to this problem is to examine a string using string methods, *before* it is converted to another data type. In the case of a positive integer, it is fair to assume that the string should contain only the digits 0-9, and should not contain any whitespaces between digits. With these assumptions, the strings "42" and " 90210" would be considered positive integers, but "-3", "8A" and "1 2 3" are not. The function below uses string methods to check if a user-inputted string passes meets these criteria.

```
import math
def get_pos_integer(low=0, high=math.inf, prompt="Enter a value: "):
   invalid = True
   while invalid:
      n = input(prompt).strip()
      if not n.isdigit():
         print("Invalid characters found, try again.")
      else:
         n = int(n)
         if n < low or n > high:
            print("Value must be between ", low, " and ", high, sep="")
         else:
            invalid = False
   return n
```

Note that we have included both `low` and `high` limits, to make the function more versatile. The first thing that happens is an `invalid` flag is set to `True`, suggesting that the string is not acceptable for conversion. The `if` block checks if the string is composed entirely of digits 0-9. If any non-digit characters are in the string, an error message is displayed and program control returns to the beginning of the loop. If the string is made entirely of digits, then the string is converted to an integer. This is safe to do, and will not cause the program to crash. Once it has been converted, the nested `if` block checks whether or not the value is between the `low` and `high` limits. If it is not, an error message is displayed. Otherwise, the `invalid` flag is set to `False`, which ends the loop. Try calling it several times to see.

This is a good way of writing such a function, as it provides the user with a lot of useful information if their input is not satisfactory. If we don't mind giving up the variety of error messages, we can make the code shorter by combining conditions using boolean operators. The code below will achieve the same goal, but will only display `Invalid value, try again` when the user enters inappropriate data. Try running it several times to convince yourself that it behaves in the same way.

```
import math
def get_pos_integer(low=0, high=math.inf, prompt="Enter a value: "):
   n = input(prompt).strip()
   while not n.isdigit() or int(n) < low or int(n) > high:
      n = input("Invalid value, try again: ").strip()
   return int(n)
```

It is important to check the characters using `isdigit` before attempting to convert `n` to an `int`. This is because Python will short-circuit the comparisons. If `n.isdigit` is `False`, then not `n.isdigit` will be `True`. Since all conditions are linked by `or`, only one conditoin must be `True` for the loop to execute. Therefore, the interpreter does not check either of the `int(n) < low` or `int(n) > high` conditions that would normally cause the program to crash on non-numeric input. Try changing the order of the conditions, and you should see an error message.

As another example of using string methods to validate user input, the function below ensures that the user enters a ten-digit phone number. The user may do this in several ways, such as (xxx) xxx-xxxx, xxx-xxx-xxxx, or simply xxxxxxxxxx. The function will read a number from the user, and ignore the characters '(', ')', '-' and spaces. If the remaining number consists of ten digits, then it will return the phone number formatted in the form (xxx) xxx-xxxx. If it contains any other characters, or if it is not ten-digits long, then the user will be prompted to enter the number again.

```
def get_phone_number(prompt="Enter your phone number: "):
   number = input(prompt)
   while True:
      new_num = ""
      for char in number:
         if char.isdigit():
            new_num += char
         elif char in "()- ":
            pass
         else:
            break
      if len(new_num) == 10:
         return "(" + new_num[:3] + ") " + new_num[3:6] + "-" + new_num[6:]
      else:
         number = input("Invalid phone number, try again: ")
```

There are a few things to point out here. The first is that the function “builds up” a new, possibly valid phone number using concatenation. If the loop encounters a valid digit, it is added to the end of the string `new_num`. If at any time the loop comes across a character that is not a digit, or is not one of the permitted special characters, then the `else` statement is executed, terminating the loop early with `break`. Should the loop complete naturally, then `new_num` will consist entirely of digits. If there are exactly ten of them, the phone number is formatted using slices and returned to the main program; otherwise, the user must enter the phone number again.

There are other ways to check whether data can be converted to certain types, such as by using **exceptions**. These are not typically covered in this course, but are not too difficult to understand if you are sufficiently motivated.