8
INPUT VALIDATION

Input validation code checks that values entered by the user, such as text from the `input()` function, are formatted correctly. For example, if you want users to enter their ages, your code shouldn't accept nonsensical answers such as negative numbers (which are outside the range of acceptable integers) or words (which are the wrong data type). Input validation can also prevent bugs or security vulnerabilities. If you implement a `withdrawFromAccount()` function that takes an argument for the amount to subtract from an account, you need to ensure the amount is a positive number. If the `withdrawFromAccount()` function subtracts a negative number from the account, the "withdrawal" will end up adding money!

Typically, we perform input validation by repeatedly asking the user for input until they enter valid text, as in the following example:

```
while True:
    print('Enter your age:')
    age = input()
    try:
        age = int(age)
    except:
        print('Please use numeric digits.')
```

```
        continue
    if age < 1:
        print('Please enter a positive number.')
        continue
    break

print(f'Your age is {age}.')
```

When you run this program, the output could look like this:

```
Enter your age:
five
Please use numeric digits.
Enter your age:
-2
Please enter a positive number.
Enter your age:
30
Your age is 30.
```

When you run this code, you'll be prompted for your age until you enter a valid one. This ensures that by the time the execution leaves the `while` loop, the `age` variable will contain a valid value that won't crash the program later on.

However, writing input validation code for every `input()` call in your program quickly becomes tedious. Also, you may miss certain cases and allow invalid input to pass through your checks. In this chapter, you'll learn how to use the third-party PyInputPlus module for input validation.

The PyInputPlus Module

PyInputPlus contains functions similar to `input()` for several kinds of data: numbers, dates, email addresses, and more. If the user ever enters invalid input, such as a badly formatted date or a number that is outside of an intended range, PyInputPlus will reprompt them for input just like our code in the previous section did. PyInputPlus also has other useful features like a limit for the number of times it reprompts users and a timeout if users are required to respond within a time limit.

PyInputPlus is not a part of the Python Standard Library, so you must install it separately using Pip. To install PyInputPlus, run `pip install --user pyinputplus` from the command line. Appendix A has complete instructions for installing third-party modules. To check if PyInputPlus installed correctly, import it in the interactive shell:

```
>>> import pyinputplus
```

If no errors appear when you import the module, it has been successfully installed.

PyInputPlus has several functions for different kinds of input:

`inputStr()` Is like the built-in `input()` function but has the general PyInputPlus features. You can also pass a custom validation function to it

`inputNum()` Ensures the user enters a number and returns an int or float, depending on if the number has a decimal point in it

`inputChoice()` Ensures the user enters one of the provided choices

`inputMenu()` Is similar to `inputChoice()`, but provides a menu with numbered or lettered options

`inputDatetime()` Ensures the user enters a date and time

`inputYesNo()` Ensures the user enters a “yes” or “no” response

`inputBool()` Is similar to `inputYesNo()`, but takes a “True” or “False” response and returns a Boolean value

`inputEmail()` Ensures the user enters a valid email address

`inputFilepath()` Ensures the user enters a valid file path and filename, and can optionally check that a file with that name exists

`inputPassword()` Is like the built-in `input()`, but displays * characters as the user types so that passwords, or other sensitive information, aren’t displayed on the screen

These functions will automatically reprompt the user for as long as they enter invalid input:

```
>>> import pyinputplus as pyip
>>> response = pyip.inputNum()
five
'five' is not a number.
42
```

```
>>> response
42
```

The `as pyip` code in the `import` statement saves us from typing `pyinputplus` each time we want to call a PyInputPlus function. Instead we can use the shorter `pyip` name. If you take a look at the example, you see that unlike `input()`, these functions return an `int` or `float` value: `42` and `3.14` instead of the strings `'42'` and `'3.14'`.

Just as you can pass a string to `input()` to provide a prompt, you can pass a string to a PyInputPlus function's `prompt` keyword argument to display a prompt:

```
>>> response = input('Enter a number: ')
Enter a number: 42
>>> response
'42'
>>> import pyinputplus as pyip
>>> response = pyip.inputInt(prompt='Enter a number: ')
Enter a number: cat
'cat' is not an integer.
Enter a number: 42
>>> response
42
```

Use Python's `help()` function to find out more about each of these functions. For example, `help(pyip.inputChoice)` displays help information for the `inputChoice()` function. Complete documentation can be found at *https://pyinputplus.readthedocs.io/*.

Unlike Python's built-in `input()`, PyInputPlus functions have several additional features for input validation, as shown in the next section.

The min, max, greaterThan, and lessThan Keyword Arguments

The `inputNum()`, `inputInt()`, and `inputFloat()` functions, which accept int and float numbers, also have `min`, `max`, `greaterThan`, and `lessThan` keyword arguments for specifying a range of valid values. For example, enter the following into the interactive shell:

```
>>> import pyinputplus as pyip
>>> response = pyip.inputNum('Enter num: ', min=4)
Enter num:3
Input must be at minimum 4.
```

```
Enter num:4
>>> response
4
>>> response = pyip.inputNum('Enter num: ', greaterThan=4)
Enter num: 4
Input must be greater than 4.
Enter num: 5
>>> response
5
>>> response = pyip.inputNum('>', min=4, lessThan=6)
Enter num: 6
Input must be less than 6.
Enter num: 3
Input must be at minimum 4.
Enter num: 4
>>> response
4
```

These keyword arguments are optional, but if supplied, the input cannot be less than the `min` argument or greater than the `max` argument (though the input can be equal to them). Also, the input must be greater than the `greaterThan` and less than the `lessThan` arguments (that is, the input cannot be equal to them).

The blank Keyword Argument

By default, blank input isn't allowed unless the `blank` keyword argument is set to `True`:

```
>>> import pyinputplus as pyip
>>> response = pyip.inputNum('Enter num: ')
Enter num:(blank input entered here)
Blank values are not allowed.
Enter num: 42
>>> response
42
>>> response = pyip.inputNum(blank=True)
(blank input entered here)
```

```
>>> response
''
```

Use `blank=True` if you'd like to make input optional so that the user doesn't need to enter anything.

The limit, timeout, and default Keyword Arguments

By default, the PyInputPlus functions will continue to ask the user for valid input forever (or for as long as the program runs). If you'd like a function to stop asking the user for input after a certain number of tries or a certain amount of time, you can use the `limit` and `timeout` keyword arguments. Pass an integer for the `limit` keyword argument to determine how many attempts a PyInputPlus function will make to receive valid input before giving up, and pass an integer for the `timeout` keyword argument to determine how many seconds the user has to enter valid input before the PyInputPlus function gives up.

If the user fails to enter valid input, these keyword arguments will cause the function to raise a `RetryLimitException` or `TimeoutException`, respectively. For example, enter the following into the interactive shell:

```
>>> import pyinputplus as pyip
>>> response = pyip.inputNum(limit=2)
blah
'blah' is not a number.
Enter num: number
'number' is not a number.
Traceback (most recent call last):
    --snip--
pyinputplus.RetryLimitException
>>> response = pyip.inputNum(timeout=10)
42 (entered after 10 seconds of waiting)
Traceback (most recent call last):
    --snip--
pyinputplus.TimeoutException
```

When you use these keyword arguments and also pass a `default` keyword argument, the function returns the default value instead of raising an exception. Enter the following into the interactive shell:

```
>>> response = pyip.inputNum(limit=2, default='N/A')
hello
'hello' is not a number.
world
'world' is not a number.
>>> response
'N/A'
```

Instead of raising `RetryLimitException`, the `inputNum()` function simply returns the string `'N/A'`.

The allowRegexes and blockRegexes Keyword Arguments

You can also use regular expressions to specify whether an input is allowed or not. The `allowRegexes` and `blockRegexes` keyword arguments take a list of regular expression strings to determine what the PyInputPlus function will accept or reject as valid input. For example, enter the following code into the interactive shell so that `inputNum()` will accept Roman numerals in addition to the usual numbers:

```
>>> import pyinputplus as pyip
>>> response = pyip.inputNum(allowRegexes=[r'(I|V|X|L|C|D|M)+', r'zero'])
XLII
>>> response
'XLII'
>>> response = pyip.inputNum(allowRegexes=[r'(i|v|x|l|c|d|m)+', r'zero'])
xlii
>>> response
'xlii'
```

Of course, this regex affects only what letters the `inputNum()` function will accept from the user; the function will still accept Roman numerals with invalid ordering such as `'XVX'` or `'MILLI'` because the `r'(I|V|X|L|C|D|M)+'` regular expression accepts those strings.

You can also specify a list of regular expression strings that a PyInputPlus function won't accept by using the `blockRegexes` keyword argument. Enter the following into the interactive shell so that `inputNum()` won't accept even numbers:

```
>>> import pyinputplus as pyip
>>> response = pyip.inputNum(blockRegexes=[r'[02468]$'])
```

```
42
This response is invalid.
44
This response is invalid.
43
>>> response
43
```

If you specify both an `allowRegexes` and `blockRegexes` argument, the allow list overrides the block list. For example, enter the following into the interactive shell, which allows `'caterpillar'` and `'category'` but blocks anything else that has the word `'cat'` in it:

```
>>> import pyinputplus as pyip
>>> response = pyip.inputStr(allowRegexes=[r'caterpillar', 'category'],
blockRegexes=[r'cat'])
cat
This response is invalid.
catastrophe
This response is invalid.
category
>>> response
'category'
```

The PyInputPlus module's functions can save you from writing tedious input validation code yourself. But there's more to the PyInputPlus module than what has been detailed here. You can examine its full documentation online at *https://pyinputplus.readthedocs.io/*.

Passing a Custom Validation Function to inputCustom()

You can write a function to perform your own custom validation logic by passing the function to `inputCustom()`. For example, say you want the user to enter a series of digits that adds up to 10. There is no `pyinputplus.inputAddsUpToTen()` function, but you can create your own function that:

- Accepts a single string argument of what the user entered
- Raises an exception if the string fails validation

- Returns `None` (or has no `return` statement) if `inputCustom()` should return the string unchanged
- Returns a non-`None` value if `inputCustom()` should return a different string from the one the user entered
- Is passed as the first argument to `inputCustom()`

For example, we can create our own `addsUpToTen()` function, and then pass it to `inputCustom()`. Note that the function call looks like `inputCustom(addsUpToTen)` and not `inputCustom(addsUpToTen())` because we are passing the `addsUpToTen()` function itself to `inputCustom()`, not calling `addsUpToTen()` and passing its return value.

```
>>> import pyinputplus as pyip
>>> def addsUpToTen(numbers):
...   numbersList = list(numbers)
...   for i, digit in enumerate(numbersList):
...     numbersList[i] = int(digit)
...   if sum(numbersList) != 10:
...     raise Exception('The digits must add up to 10, not %s.' %
(sum(numbersList)))
...   return int(numbers) # Return an int form of numbers.
...
>>> response = pyip.inputCustom(addsUpToTen) # No parentheses after
addsUpToTen here.
123
The digits must add up to 10, not 6.
1235
The digits must add up to 10, not 11.
1234
>>> response # inputStr() returned an int, not a string.
1234
>>> response = pyip.inputCustom(addsUpToTen)
hello
invalid literal for int() with base 10: 'h'
55
>>> response
```

The `inputCustom()` function also supports the general PyInputPlus features, such as the `blank`, `limit`, `timeout`, `default`, `allowRegexes`, and `blockRegexes` keyword arguments. Writing your own custom validation function is useful when it's otherwise difficult or impossible to write a regular expression for valid input, as in the "adds up to 10" example.

Project: How to Keep an Idiot Busy for Hours

Let's use PyInputPlus to create a simple program that does the following:

1. Ask the user if they'd like to know how to keep an idiot busy for hours.
2. If the user answers no, quit.
3. If the user answers yes, go to Step 1.

Of course, we don't know if the user will enter something besides "yes" or "no," so we need to perform input validation. It would also be convenient for the user to be able to enter "y" or "n" instead of the full words. PyInputPlus's `inputYesNo()` function will handle this for us and, no matter what case the user enters, return a lowercase `'yes'` or `'no'` string value.

When you run this program, it should look like the following:

```
Want to know how to keep an idiot busy for hours?
sure
'sure' is not a valid yes/no response.
Want to know how to keep an idiot busy for hours?
yes
Want to know how to keep an idiot busy for hours?
y
Want to know how to keep an idiot busy for hours?
Yes
Want to know how to keep an idiot busy for hours?
YES
Want to know how to keep an idiot busy for hours?
YES!!!!!!
'YES!!!!!!' is not a valid yes/no response.
Want to know how to keep an idiot busy for hours?
TELL ME HOW TO KEEP AN IDIOT BUSY FOR HOURS.
'TELL ME HOW TO KEEP AN IDIOT BUSY FOR HOURS.' is not a valid yes/no response.
```

```
Want to know how to keep an idiot busy for hours?
no
Thank you. Have a nice day.
```

Open a new file editor tab and save it as *idiot.py*. Then enter the following code:

```
import pyinputplus as pyip
```

This imports the PyInputPlus module. Since `pyinputplus` is a bit much to type, we'll use the name `pyip` for short.

```
while True:
    prompt = 'Want to know how to keep an idiot busy for hours?\n'
    response = pyip.inputYesNo(prompt)
```

Next, `while True:` creates an infinite loop that continues to run until it encounters a `break` statement. In this loop, we call `pyip.inputYesNo()` to ensure that this function call won't return until the user enters a valid answer.

```
    if response == 'no':
        break
```

The `pyip.inputYesNo()` call is guaranteed to only return either the string `yes` or the string `no`. If it returned `no`, then our program breaks out of the infinite loop and continues to the last line, which thanks the user:

```
print('Thank you. Have a nice day.')
```

Otherwise, the loop iterates once again.

You can also make use of the `inputYesNo()` function in non-English languages by passing `yesVal` and `noVal` keyword arguments. For example, the Spanish version of this program would have these two lines:

```
    prompt = '¿Quieres saber cómo mantener ocupado a un idiota durante horas?\n'
    response = pyip.inputYesNo(prompt, yesVal='sí', noVal='no')
    if response == 'sí':
```

Now the user can enter either `sí` or `s` (in lower- or uppercase) instead of `yes` or `y` for an affirmative answer.

Project: Multiplication Quiz

PyInputPlus's features can be useful for creating a timed multiplication quiz. By setting the `allowRegexes`, `blockRegexes`, `timeout`, and `limit` keyword argument to `pyip.inputStr()`, you can leave most of the implementation to PyInputPlus. The less code you need to write, the faster you can write your programs. Let's create a program that poses 10 multiplication problems to the user, where the valid input is the problem's correct answer. Open a new file editor tab and save the file as *multiplicationQuiz.py*.

First, we'll import `pyinputplus`, `random`, and `time`. We'll keep track of how many questions the program asks and how many correct answers the user gives with the variables `numberOfQuestions` and `correctAnswers`. A `for` loop will repeatedly pose a random multiplication problem 10 times:

```
import pyinputplus as pyip
import random, time

numberOfQuestions = 10
correctAnswers = 0
for questionNumber in range(numberOfQuestions):
```

Inside the `for` loop, the program will pick two single-digit numbers to multiply. We'll use these numbers to create a `#Q: N × N =` prompt for the user, where `Q` is the question number (1 to 10) and `N` are the two numbers to multiply.

```
    # Pick two random numbers:
    num1 = random.randint(0, 9)
    num2 = random.randint(0, 9)

    prompt = '#%s: %s x %s = ' % (questionNumber, num1, num2)
```

The `pyip.inputStr()` function will handle most of the features of this quiz program. The argument we pass for `allowRegexes` is a list with the regex string `'^%s$'`, where `%s` is replaced with the correct answer. The `^` and `%` characters ensure that the answer begins and ends with the correct number, though PyInputPlus trims any whitespace from the start and end of the user's response first just in case they inadvertently pressed the spacebar before or after their answer. The argument we pass for `blocklistRegexes` is a list with `('.*', 'Incorrect!')`. The first string in the tuple is a regex that matches every

possible string. Therefore, if the user response doesn't match the correct answer, the program will reject any other answer they provide. In that case, the `'Incorrect!'` string is displayed and the user is prompted to answer again. Additionally, passing `8` for `timeout` and `3` for `limit` will ensure that the user only has 8 seconds and 3 tries to provide a correct answer:

```
try:
    # Right answers are handled by allowRegexes.
    # Wrong answers are handled by blockRegexes, with a custom message.
    pyip.inputStr(prompt, allowRegexes=['^%s$' % (num1 * num2)],
                          blockRegexes=[('.*', 'Incorrect!')],
                          timeout=8, limit=3)
```

If the user answers after the 8-second timeout has expired, even if they answer correctly, `pyip.inputStr()` raises a `TimeoutException` exception. If the user answers incorrectly more than 3 times, it raises a `RetryLimitException` exception. Both of these exception types are in the PyInputPlus module, so `pyip.` needs to prepend them:

```
except pyip.TimeoutException:
    print('Out of time!')
except pyip.RetryLimitException:
    print('Out of tries!')
```

Remember that, just like how `else` blocks can follow an `if` or `elif` block, they can optionally follow the last `except` block. The code inside the following `else` block will run if no exception was raised in the `try` block. In our case, that means the code runs if the user entered the correct answer:

```
else:
    # This block runs if no exceptions were raised in the try block.
    print('Correct!')
    correctAnswers += 1
```

No matter which of the three messages, "Out of time!", "Out of tries!", or "Correct!", displays, let's place a 1-second pause at the end of the `for` loop to give the user time to read it. After the program has asked 10 questions and the `for` loop continues, let's show the user how many correct answers they made:

```
    time.sleep(1) # Brief pause to let user see the result.
print('Score: %s / %s' % (correctAnswers, numberOfQuestions))
```

PyInputPlus is flexible enough that you can use it in a wide variety of programs that take keyboard input from the user, as demonstrated by the programs in this chapter.

Summary

It's easy to forget to write input validation code, but without it, your programs will almost certainly have bugs. The values you expect users to enter and the values they actually enter can be completely different, and your programs need to be robust enough to handle these exceptional cases. You can use regular expressions to create your own input validation code, but for common cases, it's easier to use an existing module, such as PyInputPlus. You can import the module with `import pyinputplus as pyip` so that you can enter a shorter name when calling the module's functions.

PyInputPlus has functions for entering a variety of input, including strings, numbers, dates, yes/no, `True`/`False`, emails, and files. While `input()` always returns a string, these functions return the value in an appropriate data type. The `inputChoice()` function allow you to select one of several pre-selected options, while `inputMenu()` also adds numbers or letters for quick selection.

All of these functions have the following standard features: stripping whitespace from the sides, setting timeout and retry limits with the `timeout` and `limit` keyword arguments, and passing lists of regular expression strings to `allowRegexes` or `blockRegexes` to include or exclude particular responses. You'll no longer need to write your own tedious while loops that check for valid input and reprompt the user.

If none of the PyInputPlus module's, functions fit your needs, but you'd still like the other features that PyInputPlus provides, you can call `inputCustom()` and pass your own custom validation function for PyInputPlus to use. The documentation at *https://pyinputplus.readthedocs.io/en/latest/* has a complete listing of PyInputPlus's functions and additional features. There's far more in the PyInputPlus online documentation than what was described in this chapter. There's no use in reinventing the wheel, and learning to use this module will save you from having to write and debug code for yourself.

Now that you have expertise manipulating and validating text, it's time to learn how to read from and write to files on your computer's hard drive.

Practice Questions

1. Does PyInputPlus come with the Python Standard Library?
2. Why is PyInputPlus commonly imported with `import pyinputplus as pyip`?
3. What is the difference between `inputInt()` and `inputFloat()`?
4. How can you ensure that the user enters a whole number between `0` and `99` using PyInputPlus?
5. What is passed to the `allowRegexes` and `blockRegexes` keyword arguments?
6. What does `inputStr(limit=3)` do if blank input is entered three times?
7. What does `inputStr(limit=3, default='hello')` do if blank input is entered three times?

Practice Projects

For practice, write programs to do the following tasks.

Sandwich Maker

Write a program that asks users for their sandwich preferences. The program should use PyInputPlus to ensure that they enter valid input, such as:

- Using `inputMenu()` for a bread type: wheat, white, or sourdough.
- Using `inputMenu()` for a protein type: chicken, turkey, ham, or tofu.
- Using `inputYesNo()` to ask if they want cheese.
- If so, using `inputMenu()` to ask for a cheese type: cheddar, Swiss, or mozzarella.
- Using `inputYesNo()` to ask if they want mayo, mustard, lettuce, or tomato.
- Using `inputInt()` to ask how many sandwiches they want. Make sure this number is 1 or more.

Come up with prices for each of these options, and have your program display a total cost after the user enters their selection.

Write Your Own Multiplication Quiz

To see how much PyInputPlus is doing for you, try re-creating the multiplication quiz project on your own without importing it. This program will prompt the user with 10

multiplication questions, ranging from 0 × 0 to 9 × 9. You'll need to implement the following features:

- If the user enters the correct answer, the program displays "Correct!" for 1 second and moves on to the next question.
- The user gets three tries to enter the correct answer before the program moves on to the next question.
- Eight seconds after first displaying the question, the question is marked as incorrect even if the user enters the correct answer after the 8-second limit.

Compare your code to the code using PyInputPlus in "Project: Multiplication Quiz" on page 196.