Theory: Testing user input

21 minutes 0 / 5 problems solved

When we write programs in Python, we often want to interact with a user, for example, to ask them to enter a value to obtain a further result. We need to be very careful with that! Users may enter not what they were asked, and it can lead to unexpected errors. To prevent this, we should **test the user's input**. The idea is to take into account all scenarios and process them correctly. This topic will cover the basics of such testings.

§1. Types of values in testing user's input

Input values can be divided into three groups:

- **expected values** are correct input values, a program requires them for implementing next steps.
- **border values** when we deal with *numeric* input; they limit the range of expected inputs and may be included or not, so they could either be expected or invalid values. We mention them separately because you should always be careful with them.
- **invalid values** are incorrect inputs: they are not what we asked for.

Now let's illustrate different types of values with an example. Imagine, we ask a user to input a number:

```
your_int = int(input("Enter an integer number between 25 and 50: "))
```

So that:

1. The integers from `26` to `49` are *expected values*. They are expected from a user.
2. `25` and `50` are *border values*. In our example, we have not specified whether we want them or not, so they can be either expected or invalid values. In a real program, we will have to explain it to the user and process them accordingly.
3. Other integers, floats, or strings are *invalid values*.

In the following sections, we will discuss the ways of testing such inputs in our code.

§2. If statements for testing

Let's modify our code and read the user input step by step so that we could process every value without errors. First, we create a function that checks the given integer and prints a message if it is a correct value or not:

```
def check(x):
    if 25 < x < 50:
        print(x, "is the right number!")
    else:
        print(x, "is the wrong number!")


your_int = int(input("Enter an integer number between 25 and 50: "))
check(your_int)
```

Be careful, the border values are not included! Let's run our code several times and see what we will get with different integers as inputs:

```
# An expected value:
# Enter an integer number between 25 and 50: 45
# 45 is the right number!

# A border value:
# Enter an integer number between 25 and 50: 25
# 25 is the wrong number!

# An invalid value:

# Enter an integer number between 25 and 50: 3

# 3 is the wrong number!
```

As you can imagine, such conditional statements are not enough to test the user input. Let's see what else we can do.

§3. Try-except block to deal with exceptions

If our user enters a float or a string, the `ValueError` will occur because the `int()` function would not be able to convert the input into an integer:

```
# Enter an integer number between 25 and 50: wrong!
# Traceback (most recent call last):
#  File "main.py", line 9, in
#    your_int = int(input("Enter a number between 25 and 50: "))
# ValueError: invalid literal for int() with base 10: 'wrong!'
```

This behavior is wrong for our program! It should continue executing if an invalid value was entered. To deal with the error, we can use the `try` - `except` block. Note that we modify the lines where the program takes the input.

```
def check(x):
    if 25 < x < 50:
        print(x, "is the right number!")
    else:
        print(x, "is the wrong number!")


try:
    your_int = int(input("Enter an integer number between 25 and 50: "))

    check(your_int)

except ValueError:

    print("You entered not an integer!")
```

Now, if the user enters a float or a string, it will produce no errors:

```
# Enter an integer number between 25 and 50: wrong!
# Your input is not an integer!
```

The `while` loop can also be extremely useful for us in the task of handling the user input.

§4. While loop for continuous input request

In the previous examples, we needed to re-run the code each time to enter another value. However, when working with a user, our program should ask for the input until a correct value is entered. We can do so with the `while` loop.

In the example below, we combine `if` statements, `try-except` block, and `while` loops for multiple checking. We also consider the border values as the expected ones from now on. We should specify this in the message for a user, and process the values respectively in the code. Note that it is more convenient now to read the input inside the function.

```
def check_input():
    while True:
        try:

        your_int = int(input("Enter an integer number between 25 and 50 (inclu
sively): "))
            if 25 <= your_int <= 50:  # border values are now included
                print(your_int, "is the right number!")
                break
            else:
                print(your_int, "is the wrong number! Try again!")

        except ValueError:

            print("Your input is not an integer! Try again!")
```

Now, the program will run until a user enters what we asked them to:

```
# Enter an integer number between 25 and 50 (inclusively): I don't want to
# Your input is not an integer! Try again!
# Enter an integer number between 25 and 50 (inclusively): 3.4
# Your input is not an integer! Try again!
# Enter an integer number between 25 and 50 (inclusively): 12
# 12 is the wrong number! Try again!
# Enter an integer number between 25 and 50 (inclusively): 45
# 45 is the right number!
```

Finally, our program can process all types of inputs, and will not crash if the user behaves unexpectedly.

§5. Built-in methods for string testing

Our previous examples described integer inputs, but in a great number of situations, we need to deal with strings. Checking strings may be needed in various situations. Imagine that you are creating a program to check the password reliability. Python provides several methods that can be used for string input testing. They will allow you to check, for instance, if your password contains both integers and letters of different cases. Below we present a table with the string methods and their brief explanations.

Method	The returned value
`str.islower()`	True if there are only symbols of the lower case in the string.
`str.isupper()`	True if there are only symbols of the upper case in the string.
`str.isalpha()`	True if the string consists only of letters (upper/lower case).
`str.isdigit()`	True if the string consists only of digits.
`str.isnumeric()`	True if the string consists of digits and characters that have features of Unicode digits (their *Numeric_Type* feature is set to *Digit*, *Decimal*, or *Numeric*). For instance, the fraction `⅝` is a symbol of Unicode, meanwhile the string `"5/8"` contains three symbols. The second example will return `False` as there is a slash, whose type is not the `Numeric_Type`.
`str.isalnum()`	True if the string consists only of digits and letters (upper/lower case).

Now let's look at the example. There is a function below that takes a string and checks if it is a name; it should contain only letters, start with a capital letter, and the rest of the letters should be lowercase. These conditions are very simple and may not identify all names correctly, but you can still see how the string methods work:

```
def check_name(x):
    if x.isalpha() and x[0].isupper() and x[1:].islower():
        print("The name is", x)
    else:
        print(x, "is not a name!")


name = input("Enter a name: ")
check_name(name)
```

If any of the conditions is not met, `False` will be returned and the code will execute the `else` statement.

```
# Enter a name: Marie
# The name is Marie

# Enter a name: no
# no is not a name!

# Enter a name: Why?
# Why? is not a name!


# Enter a name: HaHa

# HaHa is not a name!
```

Take a look at the question mark in the third input. It is not a letter, so `x.isalpha()` returns `False`.

§6. Conclusion

The testing of user inputs is a pivotal step for creating a working piece of code. It allows us to process all possible inputs of a user and prevent some errors. So far, we have discussed the following:

- we have three main types of input values: the ***expected*** (correct values), the ***border*** (end-points of the expected range), and the ***invalid*** (incorrect values) ones;
- how to test inputs with the help of the `if` statements and update your testing code using the `while` loops and the `try-except` blocks;
- the methods that can be used for string testing — `str.isdigit()`, `str.isalpha()`, `str.isnumeric()`, `str.isalnum()`, `str.isupper()` and `str.islower()`.

Of course, these methods are the basic ones. For instance, you can process inputs with the help of regular expressions. You can find out more about them in our topic on regular expressions.

Report a typo

29 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

Comments (0) Hints (0) Useful links (0) Show discussion