User Input Prompts

Section 22.7

Computer programs are significantly more useful to us if they can take information provided by the user and incorporate that input into the program. We can already start to see where we clearly need user input in some of our previous examples, such as Agent J deciding whether or not he has time to wait for coffee. His user input would be things like the number of minutes he has on that particular day to wait, how badly he wants coffee that day, and how many people are currently in line ahead of him that he would have to wait behind.

So how do we capture this user input for use in our programs?

Before we get started on the *how* let's talk about how user input can be dangerous.

Being wary of user input

Providing users a way putting input into a program is a key component of what makes a program useful, but it also represents one of the biggest security risks to our program. As soon as we open our program to user input, we give the user a little bit of control. That's often exactly what we want to do – give the user the ability to control bits of the program is probably what we've designed it to do – but it's very easy to give the user *too much* control in unintended ways, and this is where security vulnerabilities often hide.

We should always approach user input with a healthy heaping of suspicion. Allow it where your program needs it, *but never fully trust it*. Assume the user might be a malicious attacker trying to exploit the system, and take precautions with the ways you allow user data to enter and travel around your program.

Why you should always use raw_input() and never input()

In a lot of other courses and books that teach Python (either 2 or 3), you might have encountered the `input()` method, which allows us to incorporate user input into a program. Both Python 2 and Python 3 have this method, but depending on which version of Python you're using it behaves differently and – most importantly – ***in Python 2 the `input()` method is extremely insecure and should never be used.***

In Python 2 we should always use the `raw_input()` method. Always.

The `raw_input()` method will interpret everything the user feeds into it as a string, no matter what characters are typed. So even if the user provides a number – like 42 – this will be interpreted as the string '42'. The same goes if the user types 'False': it gets interpreted as the string 'False' rather than the boolean value `False`.

However, if we use the `input()` method, it tries to figure out the intended type of the user input: it wants to interpret 42 as the integer 42 and 'False' as the boolean value `False`. This seems like it should be helpful, but the way it does this behind the scenes is dangerous. If we dig into the particulars of how this method works, we see it uses the very dangerous method `eval()` to figure out what type the input should be.

Why is `eval()` dangerous? Because it will evaluate what is passed to it *as code*. So a user could hypothetically type a function definition and a function call into the space we've provided for user input, and since that user input gets run through `eval()` it would actually *execute that function!* As you can see, this is a *terrible*, ***terrible idea.*** This is why we always use `raw_input()` when writing Python 2 code: we never want to send anything a user controls to an `eval()` method.

This is an important thing to remember as a programmer: you should always know how user input methods work behind the scenes so you know exactly what they're doing and how they work, to prevent creating unintentional security vulnerabilities in your code. In general, you should try and identify the known dangerous methods like `eval()` and find out where they might be used in other, seemingly innocent methods.

Python 3 and input()

If you have already encountered Python 3, you may already know that `raw_input()` doesn't exist in Python 3. That's because in Python 3, `raw_input()` was *renamed* as `input()` and behaves exactly as `raw_input()` does in Python 2. The more dangerous user input method that uses `eval()` was removed entirely from Python 3, and for good reason. However, this has made things a bit more confusing if you're jumping from Python 3 to Python 2 or back again. When in doubt, default to using `raw_input()`. The system will give you an error if you're using Python 3, where this method doesn't exist.

User input with raw_input()

Let's give the `raw_input()` method a try by having our program ask us a simple question.

```
user_name = raw_input('Hello, what is your name? ')
print('Hi ' + user_name + ', nice to meet you!')
```

When we run this code from our terminal, we get the following prompt:

```
$ python program.py
Hello, what is your name?
```

The program pauses here until we give it some user input. We type our name into the terminal after this prompt, and hit the enter key when we're done. This allows the program to continue on to line 2, with our user input in hand.

Here is our final output, from start to finish, including where we typed our name in.

```
$ python program.py
Hello, what is your name? Agent L
Hi Agent L, nice to meet you!
```

We've successfully taken in some user input, and used it to print out a nice greeting for our user with their name in it.

Let's try another example, showing how `raw_input()` always interprets our user input as a string.

```
user_coffee_input = raw_input('How many cups of coffee have you had today? ')

if user_coffee_input > 2:
    print('Wow, that\'s a lot of coffee!')
elif user_coffee_input == 0:
    print('Should we go grab a coffee? I could use one too.')
else:
    print('Sounds like the right amount of coffee to start the day.')
```

What happens when we run this program? Things go ok until after we enter our user input, then we don't get quite what we expect.

```
$ python program.py
How many cups of coffee have you had today? 0
Wow, that's a lot of coffee!
```

But wait, our program should take the 0 and run through the `elif user_coffee_input == 0` branch of our conditional and print "Should we go grab a coffee? I could use one too." but that's not what it does.

The reason why is that our user input "0" is being interpreted as a string of `"0"` rather than an integer `0`. In Python 3 we'd get a TypeError in this case where we're trying to use > to compare a string value and an integer value, but Python 2 uses different rules and thus we get our confusing output.

We can fix this using our tried and tested `int()` function in our program.

```
user_coffee_input = raw_input('How many cups of coffee have you had today? ')

user_coffee_int = int(user_coffee_input)

if user_coffee_int > 2:
    print('Wow, that\'s a lot of coffee!')
elif user_coffee_int == 0:
    print('Should we go grab a coffee? I could use one too.')
else:
    print('Sounds like the right amount of coffee to start the day.')
```

In this revised code we've added a line that transforms our user input into an integer safely, so we can now use it in our program correctly.

The output we get is below.

```
$ python program.py
How many cups of coffee have you had today? 0
Should we go grab a coffee? I could use one too.
```

Except... now what happens when the user types a string like "none" instead of a number? We'd *expect* them to type a number, but an important lesson to learn when integrating user data into our programs is to *expect the unexpected*. So let's not assume the user will type a number.

```
$ python program.py
How many cups of coffee have you had today? None
Traceback (most recent call last):
  File "program.py", line 3, in <module>
    userCoffeeInt = int(userCoffeeInput)
ValueError: invalid literal for int() with base 10: 'None'
```

Oops! We thought we had fixed a bug previously, and we sort of did... but by doing so, we introduced a new one. This isn't an unusual occurrence for a programmer: often fixing one bug reveals another. That's ok: let's keep squashing them.

Let's make sure we can deal with any type of input our user throws at us to make our program more resilient.

The `isdigit()` method can help us out here. This method allows us to check if the string the user provided can be turned into an integer. If it can't then we can provide a helpful error to our user. To do this, we're also going to refactor our code to use a couple of functions to make it easier to read and understand.

```
# Checks if the user's answer can be used by the determineReply()
# function, and if it cannot provides an error.
def get_reply(user_input):
    if user_input.isdigit():
        user_input_int = int(user_input)
        return determine_reply(user_input_int)
    else:
        return 'Sorry, I don\'t understand your answer. I was looking for a number, not
a string.'

# Determines the correct reply
def determine_reply(user_input_int):
    if user_input_int > 2:
        return 'Wow, that\'s a lot of coffee!'
    elif user_input_int == 0:
        return 'Should we go grab a coffee? I could use one too.'
    else:
        return 'Sounds like the right amount of coffee to start the day.'

# Ask for user input
user_coffee_input = raw_input('How many cups of coffee have you had today? ')

# Process the answer to get the right reply, and print that reply
reply = get_reply(user_coffee_input)
print(reply)
```

We've separated our logic into 2 function definitions. The first is called `get_reply()`, which determines if the user's input can be used in the way we want using the `isdigit()` method. If the user input can be transformed into an integer, `get_reply()` does this transform, then it returns another function with this transformed user input passed as an argument. Otherwise, if the user's string can't be turned into an integer, we return an error to be output to the screen.

The second function definition assumes an integer, and uses an if-elif-else conditional to return an appropriate reply string.

Now, after we ask for user input, we call the `get_reply()` function to sort out which of our 4 reply options we should use, and store that returned reply string to the variable `reply`, which we then print to the screen.

Time Remaining: 59 mins **Language:** Python 2 | Python 3 | C **Theme:** Dark | Light **Status:** Editor Terminal Server **Server:** Reset **Log:** Download

Steps

Step 1

Give this example code a few tries in the code editor below with different inputs to see if you can trip it up. One way it is still flawed is that you can't give it an answer like 2.5 and have it understand that user input as a float: it will see that answer as a string and give you an error, even though 2.5 can be compared to 2 using the > operator. How would you fix this?

Step 2

Currently, if this program can't understand your answer because it's a string, it gives you an error message and you have to re-run it to try again. How would you incorporate a loop into this code so that the program will ask again if it doesn't understand your answer? *Hint: this is a great use case for a while loop and a flag!*

Step 3

If the program asks you if you want to get coffee, have it get user input where if you reply with "yes" it responds with "Ok, let's go!", and if you respond with "no" it responds "Ok, see you later." And if don't respond with either "yes" or "no", it should ask you again since it didn't understand the answer you gave it.

Editor – file: ~/output.py

```
1
```

Save

Terminal – user: juZ3E0iBri

```
juZ3E0iBri@7c20dd7f3815:~$
```