Decisions

February 22, 2021

1 Making Decisions: the `if` Statement

The `if` statement is responsible for automatic decision-making.

Here's how we can pick random numbers:

[1]:
```
from random import randint
print(randint(1,10))
```

```
5
```

Each time we call `randint`, a new random value is selected. Values may repeat, but they are unpredictable and uniformly distributed (there is equal probability that any of the values are selected.

Suppose we select a value and we want to celebrate that the value is the lucky number, 8. How would we do that?

The `if` statement allows us to execute a *suite* of statements, but only if some condition holds true. Here, for example, we pick a random value, `x`, and check to see if it is 8. *If* the *condition* is met, *then* we print a message. If the condition is not met, the print statement is not executed:

[2]:
```
x = randint(1,10)
if x == 8:
    print("It's your lucky day!") # in double quotes: apostrophe in "it's"
```

When you run the cell above, it only prints the message when `x` is the value 8; otherwise, nothing happens. Experiment by running the cell over and over (press control-Enter to evaluate the cell, without moving on).

The `print` statement is the *then* part of the `if`. It forms the *suite* of statements that are executed when the condition is met.

If you want, you can choose between two alternative suites: the *then* suite and the *else* suite. Those `if` statements look like the following:

[3]:
```
x = randint(1,10)
if x == 8:
    print("You're so lucky: you picked 8!")
else:
    print("You picked",x)
```

```
You picked 3
```

Here's how we would pick two dice, print their values, and comment on pairs that come up. Notice how the indentation helps us understand which statements are part of the suite of a particular `if`.

[4]:
```
die1 = randint(1,6)
die2 = randint(1,6)
print('You rolled a',die1,'and a',die2)
if die1 == die2:
    if die1 == 6:                # dice are the same, and they're sixes
        print('Double boxcars!')
    else:                        # dice are the same, but not sixes
        print("It's a pair!")
print('The dice sum to',die1+die2)
```

```
You rolled a 1 and a 2
The dice sum to 3
```

The `else` part, if provided, is executed if the condition fails.

Suppose we roll a die and want to print a message if the value is prime. (The possible dice rolls that are prime would be 2, 3, or 5.) Here's how we do that:

[5]:
```
x = randint(1,6)
print('You rolled a',x)
if (x == 2) or (x == 3) or (x == 5):
    print("That's a prime.")
else:
    print("That's not a prime.")
```

```
You rolled a 3
That's a prime.
```

It always prints some message, but the computer *decides* which message is printed.

By the way, we could capture the prime test in a variable. This code works exactly the same as the above, but our intention is a little clearer.

[6]:
```
x = randint(1,6)
print('You rolled a',x)
isPrime = (x == 2) or (x == 3) or (x == 5)
if isPrime:
    print("That's a prime.")
else:
    print("That's not a prime.")
```

```
You rolled a 5
That's a prime.
```

When we have multiple choices, we could nest the statments, as follows:

```
cookies = randint(0,4)  if cookies == 0:     print('You have no cookies.')  else:
if cookies == 1:         print('You have one cookie.')     else:         if
cookies == 2:             print('You have a couple of cookies.')         else:
print('You have a bunch of cookies.')
```

Notice how the nesting of the `if` statements indicates, for any `print` statement, what conditions succeeded or failed. While this is perfectly legal, the selection from a large number of choices can lead to really deep indentation.

A nice alternative is the use of the `elif` statement, which replaces the `else...if` syntax that leads to deep indentation. Here's what it looks like:

[7]:
```
cookies = randint(0,4)
if cookies == 0:
   print('You have no cookies.')
elif cookies == 1:
   print('You have one cookie.')
elif cookies == 2:
   print('You have a couple of cookies.')
else:
   print('You have a bunch of cookies.')
```

```
You have a couple of cookies.
```

The resulting expression is (1) much easier to read, and (2) can be extended to include as many tests as necessary to conver all the possibilities.

Next, we think about how to resolve fights over remaindered cookies.

1.1 Rock, Paper, Scissors

Here's a popular game: Rock, Paper, Scissors. Two people count down and then "shoot" out a hand in one of three poses: showing two fingers (meaning *scissors*), flat (meaning *paper*), or as a fist (meaning *rock*). The exposed hands are then compared, according to the following rules: 1. If the two hand are posed the same way, it's a draw. 2. If rock and paper are exposed, *paper wins*, and we say "paper covers rock." 3. If rock and scissors are exposed, *rock wins*, and we say "rock breaks scissors." 4. If paper and scissors are exposed, *scissors wins*, and we say "scissors cuts paper". We can write code to play a round of Rock, Paper, Scissors.

First, let's randomly assign a variable, `alice` one of three strings, `"rock"`, `"paper"`, or `"scissors"`. There are three choices, so we'll pick one of three integers and then assign alice based on the integer chosen:

[8]:
```
poseNumber = randint(1,3) # pick one of three configurations
if poseNumber == 1:
    alice = "rock"
elif poseNumber == 2:
    alice = "paper"
else: # poseNumber must be 3
    alice = "scissors"
# always true, here: alice has a value
```

[9]:
```
alice
```

[9]:
```
'scissors'
```

We can now do the same thing for `bob`:

[10]:
```
poseNumber = randint(1,3) # pick one of three configurations
if poseNumber == 1:
    bob = "rock"
elif poseNumber == 2:
    bob = "paper"
else: # poseNumber must be 3
    bob = "scissors"
# always true, here: bob has a value
```

[11]:
```
bob
```

[11]:
```
'paper'
```

Now, we can think about how to evaluate whether `alice` or `bob` wins, or if it's a draw. When we write this code, we'll think about it slightly differently: we'll focus on winning situations for a player:

[12]:
```
if alice == bob:
    print("Draw!")
elif (((alice == "paper") and (bob == "rock"))
      or ((alice == "rock") and (bob == "scissors"))
      or ((alice == "scissors") and (bob == "paper"))):
    print("Alice wins!")
else: # Bob *must* be a winner:
    print("Bob wins!")
```

```
Alice wins!
```

Casting the various rule tests in this manner helps to reduce the complexity of the if statements. Notice, by the way, that at the end of each line of the `elif` condition, the first parenthesis remains unmatched. Since Python prefers to have each statement on a single line, we have to make the statement incomplete, by leaving a parenthesis unmatched, to force Python to read more lines.

Actually, most of the parentheses are unnecessary. We used parens to explicitly determine the order of evaluation, but these boolean expressions have precedence, just like any other math formula. Boolean operators are very low priority, with `or` (and addition-like operation) lower than `and` (a multiplication-like operation). Here's equivalent code:

[13]:
```
shePersisted = alice == "paper" and bob == "rock" or alice == "rock" and bob ==␣
↪"scissors" or alice == "scissors" and bob == "paper"
if shePersisted:
    print("Alice gets the cookie.")
else:
```

```
    print("No cookie yet.")
```

```
Alice gets the cookie.
```