Exercise 3

In this exercise, you'll put to work what you have learned about booleans and conditionals.

To get started, **run the setup code below** before writing your own code (and if you leave this notebook and come back later, don't forget to run the setup code again).

In [1]:
```
# Run this cell to setup the exercise
import sys
from pathlib import Path
learntools_dir = Path().absolute().parents[1]
sys.path.append(str(learntools_dir))
from learntools.core import binder; binder.bind(globals())
from learntools.python.ex3 import *
print('Setup complete.')
```

```
Setup complete.
```

1.

Many programming languages have `sign` available as a built-in function. Python doesn't, but we can define our own!

In the cell below, define a function called `sign` which takes a numerical argument and returns -1 if it's negative, 1 if it's positive, and 0 if it's 0.

In [2]:
```
# Your code goes here. Define a function called 'sign'
def sign(x):
    if x > 0:
        y = 1
    elif x < 0:
        y = -1
    else:
        y = 0
    return y

# Check your answer
q1.check()
```

Correct

In [3]:
```
q1.solution()
```

Solution:

```
def sign(x):
    if x > 0:
        return 1
    elif x < 0:
        return -1
    else:
        return 0
```

2.

We've decided to add "logging" to our `to_smash` function from the previous exercise.

In [4]:
```
def to_smash(total_candies):
    """Return the number of leftover candies that must be smashed after dist
    the given number of candies evenly between 3 friends.

    >>> to_smash(91)
    1
    """
    print("Splitting", total_candies, "candies")
    return total_candies % 3

to_smash(91)
```

```
Splitting 91 candies
```

Out[4]: 1

What happens if we call it with `total_candies = 1`?

In [5]:
```
to_smash(1)
```

```
Splitting 1 candies
```

Out[5]: 1

That isn't great grammar!

Modify the definition in the cell below to correct the grammar of our print statement. (If there's only one candy, we should use the singular "candy" instead of the plural "candies")

In [6]:
```
def to_smash(total_candies):
    """Return the number of leftover candies that must be smashed after dist
    the given number of candies evenly between 3 friends.

    >>> to_smash(91)
```

```
    if total_candies > 1:
        print("Splitting", total_candies, "candies")
    else:
        print("Splitting", total_candies, "candy")
    return total_candies % 3

to_smash(91)
to_smash(1)
```

```
Splitting 91 candies
Splitting 1 candy
```

Out[6]: 1

To get credit for completing this problem, and to see the official answer, run the code cell below.

In [ ]:
```
# Check your answer
q2.solution()
```

3. 🌶️

In the tutorial, we talked about deciding whether we're prepared for the weather. I said that I'm safe from today's weather if...

- I have an umbrella...
- or if the rain isn't too heavy and I have a hood...
- otherwise, I'm still fine unless it's raining *and* it's a workday

The function below uses our first attempt at turning this logic into a Python expression. I claimed that there was a bug in that code. Can you find it?

To prove that `prepared_for_weather` is buggy, come up with a set of inputs where either:

- the function returns `False` (but should have returned `True`), or
- the function returned `True` (but should have returned `False`).

To get credit for completing this question, your code should return a Correct result.

In [10]:
```
def prepared_for_weather(have_umbrella, rain_level, have_hood, is_workday):
    # Don't change this code. Our goal is just to find the bug, not fix it!
    return have_umbrella or (rain_level < 5 and have_hood) or not (rain_leve

# Change the values of these inputs so they represent a case where prepared_
# returns the wrong answer.
have_umbrella = True
rain_level = 0.0
have_hood = True
is_workday = True
```

```
# Check what the function returns given the current values of the variables
actual = prepared_for_weather(have_umbrella, rain_level, have_hood, is_worko
print(actual)

# Check your answer
q3.check()
```

```
True
```

Incorrect: Given `have_umbrella=True`, `rain_level=0.0`, `have_hood=True`, `is_workday=True`, `prepared_for_weather` returned `True`. But I think that's correct. (We want inputs that lead to an incorrect result from `prepared_for_weather`.)

In [12]:
```
q3.hint()

q3.solution()
```

Hint: Take a look at how we fixed our original expression in the main lesson. We added parentheses around certain subexpressions. The bug in this code is caused by Python evaluating certain operations in the "wrong" order.

Solution: One example of a failing test case is:

have_umbrella = **False**
rain_level = 0.0
have_hood = **False**
is_workday = **False**
Clearly we're prepared for the weather in this case. It's not raining. Not only that, it's not a workday, so we don't even need to leave the house! But our function will return False on these inputs.

The key problem is that Python implictly parenthesizes the last part as:

(**not** (rain_level > 0)) **and** is_workday
Whereas what we were trying to express would look more like:

not (rain_level > 0 **and** is_workday)

4.

The function `is_negative` below is implemented correctly - it returns True if the given number is negative and False otherwise.

However, it's more verbose than it needs to be. We can actually reduce the number of lines of code in this function by *75%* while keeping the same behaviour.

See if you can come up with an equivalent body that uses just **one line** of code, and put it in the function `concise_is_negative`. (HINT: you don't even need Python's ternary syntax)

In [13]:
```
def is_negative(number):
    if number < 0:
        return True
    else:
        return False

def concise_is_negative(number):
    return number < 0 # Your code goes here (try to keep it to one line!)

# Check your answer
q4.check()
```

Correct

In [15]:
```
q4.hint()

# q4.solution()
```

Hint: If the value of the expression `number < 0` is `True`, then we return `True`. If it's `False`, then we return `False` ...

5a.

The boolean variables `ketchup`, `mustard` and `onion` represent whether a customer wants a particular topping on their hot dog. We want to implement a number of boolean functions that correspond to some yes-or-no questions about the customer's order. For example:

In [16]:
```
def onionless(ketchup, mustard, onion):
    """Return whether the customer doesn't want onions.
    """
    return not onion
```

```
def all_toppings(ketchup, mustard, onion):
    """Return whether the customer wants "the works" (all 3 toppings)
```

```
    """
    return ketchup and mustard and onion

# Check your answer
q5.a.check()
```

Correct

In [ ]:
```
# q5.a.hint()

# q5.a.solution()
```

5b.

For the next function, fill in the body to match the English description in the docstring.

In [21]:
```
def wants_plain_hotdog(ketchup, mustard, onion):
    """Return whether the customer wants a plain hot dog with no toppings.
    """
    return not ketchup and not mustard and not onion

# Check your answer
q5.b.check()
```

Correct:

One solution looks like:

return not ketchup **and not** mustard **and not** onion

We can also "factor out" the nots to get:

return not (ketchup **or** mustard **or** onion)

In [ ]:
```
# q5.b.hint()

# q5.b.solution()
```

5c.

You know what to do: for the next function, fill in the body to match the English description in the docstring.

In [ ]:
```
def exactly_one_sauce(ketchup, mustard, onion):
    """Return whether the customer wants either ketchup or mustard, but not
    (You may be familiar with this operation under the name "exclusive or")
    """
    pass
```

```
# Check your answer
q5.c.check()
```

In [ ]:
```
# q5.c.hint()

# q5.c.solution()
```

6. 🌶️

We've seen that calling `bool()` on an integer returns `False` if it's equal to 0 and `True` otherwise. What happens if we call `int()` on a bool? Try it out in the notebook cell below.

Can you take advantage of this to write a succinct function that corresponds to the English sentence "does the customer want exactly one topping?"?

In [ ]:
```
def exactly_one_topping(ketchup, mustard, onion):
    """Return whether the customer wants exactly one of the three available
    on their hot dog.
    """
    pass

# Check your answer
q6.check()
```

In [ ]:
```
# q6.hint()

# q6.solution()
```

7. 🌶️ (Optional)

In this problem we'll be working with a simplified version of blackjack (aka twenty-one). In this version there is one player (who you'll control) and a dealer. Play proceeds as follows:

- The player is dealt two face-up cards. The dealer is dealt one face-up card.
- The player may ask to be dealt another card ('hit') as many times as they wish. If the sum of their cards exceeds 21, they lose the round immediately.
- The dealer then deals additional cards to himself until either:
  - the sum of the dealer's cards exceeds 21, in which case the player wins the round
  - the sum of the dealer's cards is greater than or equal to 17. If the player's total is greater than the dealer's, the player wins. Otherwise, the dealer wins (even in case of a tie).

When calculating the sum of cards, Jack, Queen, and King count for 10. Aces can count as 1 or 11 (when referring to a player's "total" above, we mean the largest total that can be made without exceeding 21. So e.g. A+8 = 19, A+8+8 = 17)

For this problem, you'll write a function representing the player's decision-making strategy in this game. We've provided a very unintelligent implementation below:

In [ ]:
```
def should_hit(dealer_total, player_total, player_low_aces, player_high_aces
    """Return True if the player should hit (request another card) given the
    state, or False if the player should stay.
    When calculating a hand's total value, we count aces as "high" (with val
    doesn't bring the total above 21, otherwise we count them as low (with v
    For example, if the player's hand is {A, A, A, 7}, we will count it as 1
    and therefore set player_total=20, player_low_aces=2, player_high_aces=1
    """
    return False
```

This very conservative agent *always* sticks with the hand of two cards that they're dealt.

We'll be simulating games between your player agent and our own dealer agent by calling your function.

Try running the function below to see an example of a simulated game:

In [ ]:
```
q7.simulate_one_game()
```

The real test of your agent's mettle is their average win rate over many games. Try calling the function below to simulate 50000 games of blackjack (it may take a couple seconds):

In [ ]:
```
q7.simulate(n_games=50000)
```

Our dumb agent that completely ignores the game state still manages to win shockingly often!

Try adding some more smarts to the `should_hit` function and see how it affects the results.

In [ ]:
```
def should_hit(dealer_total, player_total, player_low_aces, player_high_aces
    """Return True if the player should hit (request another card) given the
    state, or False if the player should stay.
    When calculating a hand's total value, we count aces as "high" (with val
    doesn't bring the total above 21, otherwise we count them as low (with v
    For example, if the player's hand is {A, A, A, 7}, we will count it as 1
    and therefore set player_total=20, player_low_aces=2, player_high_aces=1
    """
    return False

q7.simulate(n_games=50000)
```

Keep Going

Learn about **lists and tuples** to handle multiple items of data in a systematic way.
