CS051A

INTRO TO COMPUTER SCIENCE WITH TOPICS IN AI
7: Boolean variables, aliasing, and parameter passing

Alexandra Papoutsaki

she/her/hers

Lectures

Zilong Ye

he/him/his

Labs

Lecture 7: Boolean variables, aliasing, and parameter passing

▶ **Python function calls**

▶ Booleans

▶ Sequences

▶ Aliasing

▶ Parameter Passing

What happens when a function gets called?

▸ Let's look at this simple file `function-calls.py`

▸ We can walkthrough what happens when we make a call to `add_then_double`, e.g., `add_then_double(10, 20)`

▸ 1. The arguments to the function (`10` and `20`) are evaluated.

  ▸ They could have been more complicated expressions, e.g., `add_then_double(5+5, 40/2)`

▸ 2. The function is then called and the parameters are associated with the arguments that are passed in. Think of it like saying `a = 10` and `b = 20`.

  ▸ We now have two variables in the function that have the values of the arguments.

  ▸ The only tricky thing is that these variables are unique to this function, so even if they're defined globally (outside any function), they will be different than the global variables.

What happens when a function gets called? (cont'd)

▸ 3. The function body for `add_then_double` is executed one statement at a time.

  ▸ This will in turn call `add(a, b)`. Like any other function call, we:

    ▸ evaluate the arguments, in this case, we evaluated a, which gives us `10`, and b, which gives us `20`,

    ▸ we then pass these values to `add` and associate them with parameters `num1` and `num2`.

    ▸ `add` is only one line of code, it adds the number and then returns that value.

      ▸ `num1 + num2` gives us `30` (`10 + 20`)

    ▸ the `return` statement means wherever this function was called, that is the value that call should represent.

      ▸ the call `add(a, b)` will then represent the value `30`

  ▸ We then assign what `add(a,b)` returns to the variable `added`.

  ▸ `double` is called with what was in the variable `added` (`30`) and following the same logic as above gives us back `60`

  ▸ `60` is then associated with the variable `doubled`.

  ▸ Finally, `add_then_double` returns `doubled`, which has the value `60`.

What happens when a function gets called? (cont'd)

▸ If we called `add_then_double` from the Python console/shell, we will see the value 60 echoed, not because it was printed out but because that call (`add_then_double(10, 20)`) now represents the value 60.

▸ We could just have easily done other things with it, e.g.,

```
>>> add_then_double(10,20)
60
>>> add_then_double(10,20) * 2
120
>>> result = add_then_double(10,20)
>>> result
60
>>> add_then_double(add_then_double(10,20),4)
128
```

Lecture 7: Boolean variables, aliasing, and parameter passing

▸ Python function calls

▸ **Booleans**

▸ Sequences

▸ Aliasing

▸ Parameter Passing

Practice time

▸ Write a function called `contains` that takes a list and an item and returns `True` if the item is in the list, `False` otherwise, i.e. `def contains(some_list, value):`

▸ Look at the `contains` function in `contains.py` file

  ▸ Loops through each value in the list.

  ▸ If it finds one that is equal to the item we're looking for, then return `True`

    ▸ Remember that `return` immediately exits the function. This will stop the loop and then give back True from the function.

  ▸ if we make it through the entire list without finding one that's equal, then the loop will finish and we'll return `False`.

    ▸ This is similar to the `isprime` function we saw before in `while.py` code

▸ The `contains_alternative` function in `contains.py` code is another way of accomplishing this.

  ▸ We use a variable `found` to keep track of whether or not we've found the item and we initialize to `False`.

  ▸ We then loop through all of the values of the list.

  ▸ If we find the item we're looking for, we set the variable `found` to `True`

    ▸ it's fine to set `found` to `True` again if the item exists multiple times in the list.

  ▸ After the loop finishes, we return `found`

    ▸ If we didn't find any, it will still be `False`, otherwise, it will have been set to `True`.

Built-in contains functionality

▸ Python has a built-in contains functionality in the form of the `in` keyword that works for lists, strings, and tuples (i.e. sequences).

▸ Notice that for lists, the item has to be in the list as one of the individual items, we cannot check for a slice.

▸ But this is not a problem in strings. We can check whether a string contains an entire substring not just a single character.

```
>>> 2 in [1, 2, 3]
True
>>> 5 in [1, 2, 3]
False
>>> "banana" in [1, 2, 3]
False
>>> [1, 2] in [1, 2, 3]
False
>>> [1, 2] in [[1, 2], 2, 3]
True
>>> "a" in "banana"
True
>>> "ana" in "banana"
True
```

`number-game.py`

▸ Picks a random number between 1 and 20 and keeps prompting the user to guess it until they get it right.

  ▸ It provides hints as to whether they need to guess higher or lower.

▸ How can we implement this game?

  ▸ We can pick a random guess that the user will have to guess.

  ▸ As long as (`while`) the user hasn't guessed the right answer:

    ▸ Ask the user for their guess.

  ▸ If it's the right answer, print out "Good job!" and somehow indicate that we're done looping.

  ▸ Otherwise, if the guess is too low print out ("A bit higher").

  ▸ Otherwise (i.e. the guess must be too high), print out ("A bit lower").

`number_guessing_game` function in `number-game.py`

‣ Uses a variable called `correct` and initially sets it to `False.`

‣ Repeats the same steps while `correct` remains `False (while not correct:).`

    ‣ We could have also use a variable like `still_guessing` and set it to True and then had `while still_guessing:`

‣ When the user guesses the right number, we set `correct` to True and therefore exit the loop.

‣ Notice that there are other ways of writing this function, e.g., `number_guessing_game2`

Lecture 7: Boolean variables, aliasing, and parameter passing

▸ Python function calls

▸ Booleans

▸ **Sequences**

▸ Aliasing

▸ Parameter Passing

A few more sequence operators

▸ We've seen + for concatenating strings. It also works on other sequences!

▸ What do you think sequence*x does?

   ▸ Repeats the sequence x times

```
>>> "banana" + " split"
'banana split'
>>> [1, 2, 3] + [4, 5]
[1, 2, 3, 4, 5]
>>> (1, 2, 3) + (4, 5)
(1, 2, 3, 4, 5)
>>> "*" * 10
'**********'
>>> "I'm " + "very " * 4 + "excited"
"I'm very very very very excited"
>>> [0] * 10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> [1, 2, 3] * 2
[1, 2, 3, 1, 2, 3]
```

Lecture 7: Boolean variables, aliasing, and parameter passing

▸ Python function calls

▸ Booleans

▸ Sequences

▸ **Aliasing**

▸ Parameter Passing

Assignment (i.e. =)

▸ when we say x = y , what we're
   actually saying is let x reference the
   same thing that y references.

▸ x and y are still separate variables.

▸ If we say x = z that does NOT change
   the value of y .

```
>>> x = 1
>>> y = 2
>>> z = 3
>>> x = y
>>> x
2
>>> y
2
>>> z
3
>>> x = z
>>> x
3
>>> y
2
>>> z
3
```

References

▸ Objects reside in memory. Anytime we create an `int`, `float`, `bool`, `str`, `list`, `tuple` it is allocated in memory.

▸ Variables are references to objects in memory. A variable does NOT hold the value itself, but it is a references to where that value resides.

    ▸ Think of variables like your hands. You can point them at things and then ask questions:

        ▸ what is the name of the thing that my right hand points to?

        ▸ for the thing that my left hand points to, do something?

    ▸ You can point them to new things (i.e. assignment)

    ▸ You can point point them to the same thing (i.e. assignment one to the other) but they are separate things!

    ▸ Similarly, variables are separate things. They can reference the same thing, but they are not the same thing.

▸ When you ask for the value of a variable, it's actually getting the value of the things it references in memory

Objects in memory

▸ If an object is mutable (i.e. it has methods that change the object), then we have to be careful when we have multiple things referencing the same object (this is called aliasing).

```
>>> x = [1, 2, 3, 4, 5]
>>> y = x
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5]
```

▸ what does this look like in memory?

  ▸ There is one list object. Both x and y are references to that same object.

Aliasing

▸ What happens when I call a method that changes/mutates the object?

▸ x and y are references to the same object! Statements that mutate the object that are done on either variable will affect this object.

```
>>> y.reverse()
>>> x
[5, 4, 3, 2, 1]
>>> y
[5, 4, 3, 2, 1]

>>> x[0] = 0
>>> x
[0, 4, 3, 2, 1]
>>> y
[0, 4, 3, 2, 1]
>>> y[0] = 15
>>> x
[15, 4, 3, 2, 1]
>>> y
[15, 4, 3, 2, 1]
```

List equality

▶ == compares whether all corresponding elements of two lists are equal.

▶ is checks whether two references point to the same memory location

```
>>> list1 = [1, 2, 3, 4]
>>> list2 = [1, 2, 3, 4]
>>> list3 = list2
>>> list1 == list2
True
>>> list2 == list3
True
>>> list1 == list3
True
>>> list1 is list2
False
>>> list1 is list3
False
>>> list2 is list3
True
```

ints/floats/bools/strings

▸   Why hasn't aliasing come up with ints/floats/bools/strings if they are objects?

```
>>> x = 10
>>> y = 10
```

▸   what does the memory picture look like?

    ▸   We just have one int object!

    ▸   x and y are both references to that one int object

▸   int/float/bool/str types are not mutable!

    ▸   There is no way for us to change the underlying object. Therefore, even though two variables reference the same int, it's as if they referenced separate ints.

Practice time

▸ Draw the memory picture for the following:

```
>>> x = 10
>>> y = x
>>> y = 15
>>> x
10
>>> y
15
```

▸ We now have 2 objects (10 and 15). Just like with lists, changing what y references does not affect x.

Lecture 7: Boolean variables, aliasing, and parameter passing

▸ Python function calls

▸ Booleans

▸ Sequences

▸ Aliasing

▸ **Parameter Passing**

Parameter passing

▸ When we define a function's parameters we are defining the "formal parameters".

```
>>> def my_function(a, b):
...     return a + b
```

    ▸ *a* and *b* are formal parameters. Until the function is actually called, they don't represent actual values.

    ▸ When we call a function, the values that we give to the function are called the "actual parameters" or arguments.

```
>>> x = 10
>>> y = 20
>>> my_function(x, y)
30
```

▸ The values 10 and 20 are the actual parameters (or similarly, x and y become the arguments)

When a function is called the following happens

▸ The values of the arguments are determined (we evaluate the expression representing each parameter).

   ▸ The value is just an object (could be an `int`, `float`, `str`, etc).

▸ These values are then "bound" or assigned to the formal parameters

   ▸ It's very similar to assignment.

      ▸ The formal parameters represent new variables.

      ▸ the formal parameters will then REFERENCE the same objects as those passed in through the arguments/actual parameters.

Referencing

▸ Let's consider the picture for the following code.

```
>>> x = 10
>>> y = 20
>>> my_function(x, y)
30
```

▸ x and y are both references to `int` objects.

▸ When we call `my_function`, the formal parameters *a* and *b*, will represent two new variables. These variables will reference the same thing as their actual parameters.

  ▸ Think of it like running the statements: *a* = x and *b* = y.

  ▸ *a* will reference the same thing as x

  ▸ *b* will reference the same thing as y

(Im)mutability and referencing

▸ For immutable objects, this whole story doesn't really matter. They could be references to the same thing or copies but if we can't mutate the object, the behavior is the same.

▸ How does this change for mutable objects, such as lists?

  ▸ x was a variable that references a list object

  ▸ When the function was called, the formal parameter numbers will represent a new variable.

  ▸ numbers will reference the same thing as x.

  ▸ Think of it like running the statement

    ▸ numbers = x

▸ Because lists are mutable and since the formal parameter references the same object as what was passed in, changes made to the object referenced by numbers will also be seen in x.

```
>>> def changer(numbers):
...     numbers[0] = 100
...
...
... def no_changer(numbers):
...     numbers = [0] * 5
...
>>> x = [1, 2, 3, 4, 5]
>>> changer(x)
>>> x
[100, 2, 3, 4, 5]
```

(Im)mutability and referencing

▸ Notice, however, that operations that do not change/mutate the object will NOT be seen outside the function

  ▸ In this case, we're assigning numbers to some new object (list).

  ▸ We can't change what x references!

  ▸ x and numbers will no longer reference the same object.

    ▸ Any changes to numbers after this will not affect x.

▸ Variable assignment and parameter passing is done based on the references (i.e. a shallow copy) rather than a deep copy of the whole object.

  ▸ Performance (it takes work to copy the object).

  ▸ Often, we just want the value and don't need to mutate the underlying object.

```
>>> def changer(numbers):
...     numbers[0] = 100
...
...
... def no_changer(numbers):
...     numbers = [0] * 5
...
>>> x = [1, 2, 3, 4, 5]
>>> changer(x)
>>> x
[100, 2, 3, 4, 5]
>>> no_changer(x)
>>> x
[100, 2, 3, 4, 5]
```

Resources

▸ Textbook: Chapters 6

▸ function-calls.py

▸ contains.py

▸ number-game.py

Homework

▸ Assignment 3 (in progress)