Lab 03: Control Flow and I/O

Contents

Objectives

By the end of this lab you will understand:

- How to execute different branches of code
- Conditional looping

By the end of this lab you will be able to:

- Write and evaluate conditional statements
- Read from and write to files
- Interact with a user

1 Boolean Operators

Up until now, the only time we have seen the `bool` type is as the output of the `isinstance` function, and as the output of the `in` operator we saw last week. In fact, this type is one of the most important ones in programming languages, as it allows us to execute different code branches depending on the different concrete values we are given. Since we don't usually know the values we expect in advance, this provides a degree of flexibility that we would not be able to achieve otherwise.

Just as we have seen basic numeric operators such as `+` and `*`, programming languages have many built-in boolean operators as well. For numeric values, these are often used to compare them, and the boolean acts as the answer to a question: e.g., is the value of this variable greater than 5? Is it equal to 10? The most common operators are (1) `<`, (2) `>`, (3) `<=`, (4) `>=`, (5) `==`, and (6) `!=`. The first four of these should be familiar from the mathematics you did in school, and test whether the value on the left-hand side is, respectively, (1) less than, (2) greater than, (3) less than or equal to, and (4) greater than or equal to the value on the right-hand side. The fifth one tests whether or not the values are equal (two equal signs are needed to distinguish it from the assignment operator), and the final one tests whether or not the values are not equal.

Task 1. In the interactive interpreter, spend 5-10 minutes using numeric values to play around with these operators until you feel comfortable with them. As always, be sure to expand past simple examples (e.g., `4 < 5`) to also incorporate more complex expressions (e.g., `4 + 4554 - 414 >= 341 + 441*2`).

Solution 1.

```
>>> 4 < 5
True
>>> 4 + 4554 - 414 >= 341 + 441*2
True
>>> x = 23
>>> y = 25
>>> x != y
True
>>> x == y
False
>>> y = y - 2
>>> x == y
True
>>> x + y - 23*2 > 0
False
```

It turns out that these operators can be used to compare non-numeric types as well; e.g., `[] < [1,2]` returns `True`, and `'abc' < 'ab'` returns `False`. Feel free to play around with the operators using these additional types as well, although we will not need to do so for now.

To combine these boolean operators, we can use the built-in logical connectives of `or`, `and`, and `not`. The first two of these are *binary*, meaning they take in a left- and right-hand side and evaluate the expression according to the following *truth tables* (where 0 is used to represent `False` and 1 is used to represent `True`):

or	0	1
0	0	1
1	1	1

and	0	1
0	0	0
1	0	1

So, an `and` expression evaluates to `True` only if all of its arguments evaluate to `True`, whereas an `or` expression evaluates to `True` as long as at least one of its arguments evaluates to `True`.

The `not` operator, in contrast, is *unary*, meaning it acts on a single value. It reverses the value of the boolean it is given, so `not True` evaluates to `False` and `not False` evaluates to `True`.

Task 2. Continue to play around with boolean operators in the interactive interpreter for another 5-10 minutes, but incorporate them into more complex expressions using the additional logical operators.

Solution 2.

```
>>> x = 23
>>> y = 25
>>> x != y and x <= y
True
>>> x == y or not x > y
True
```

2 If Statements

The `if` statement allows you to achieve *conditional* execution, meaning you can provide statements that are executed only if some expression evaluates to `True`. A basic `if` consists of two parts: the *condition* and the *if-clause*. The condition is a boolean expression, and the if-clause is a series of statements that are run if the condition evaluates to `True`. Each statement in the if-clause must be indented one level in from the statement of the condition, as in the following example:

```
if x == 1:
    print 'x is 1'
```

Task 3. Write 2-3 simple `if` statements, either in the interactive interpreter or in a file that you then execute. Experiment with including compound boolean expressions (i.e., expressions with multiple parts) in the condition, and multiple statements in the if-clause.

Solution 3.

```
if x == 5:
    a = 1
    print a
    x = 6

if x < 5 or a == 1:
    print 'hi!'
```

Task 4. Write an `if` statement that, given two numeric variables x and y, swaps their values if x is less than y. Put this in a file `swap.py`.

Solution 4. A fully commented and tested solution is in `swap.py`. The basic code is as follows:

```
if x < y:
    tmp = x
    x = y
    y = tmp
```

In many uses of `if`, you want to execute one set of statements if the condition is true, and another set of statements if not. For that, there is an accompanying `else` clause. This clause follows the set of statements in the if-clause, as in the following example:

```
if x == 1:
    print 'x is 1'
else:
    print 'x is not 1'
```

Python has an additional operator, `elif` (short for 'else if'), that allows you to check for additional conditions; effectively, it splits the execution up according to several conditions, as opposed to just two. For example:

```
if x == 1:
    print 'x is 1'
elif x == 2:
    print 'x is 2'
elif x == 3:
    print 'x is 3'
else:
    print 'x is greater than 3'
```

Finally, Python has a special *ternary* operator of the form `a if c else b`. This succinct statement executes the code in `a` if condition `c` returns `True`, and otherwise executes the code in `b`. For example, if `x = 5` and `y = 7` then `z = x if x > y else y` will result in `z` taking on the value 7.

3 While Loops

Unlike `for` loops, which are designed to iterate through sequences such as strings and lists, `while` loops are designed to execute a certain set of statements as long as (or *while*) a given condition is true. As such, a `while` loop consists of two parts: the *condition* and the *body*. Like an `if` statement, Python first evaluates the condition when executing a loop, and if it evaluates to `False` then it does not execute the body, and instead skips ahead to any code after the loop. Unlike an `if` statement, if the value is `True` then Python not only executes the body once (as in an `if` statement) but goes back at the end to test the loop condition again. If it still evaluates to `True` then Python executes the body again, and this continues until the condition evaluates to `False`. This means the following loop would run forever:[1]

```
x = 1
while True:
    print x
    x = x + 1
```

In contrast, in the following example the body of the loop is executed exactly ten times before *terminating* (i.e., reaching the terminal case in which the condition evaluates to `False`):

```
x = 10
while x > 0:
    x = x - 1
```

In addition to the simple examples above, as we saw last week with `for` loops, loops can be used to iterate through a list. This means that we can take each item in a list and perform some operation on it.

Task 5. Mimic the behavior of a `for` loop by creating a file `while_list.py` and writing in it a `while` loop that goes through a numeric list `mylist` and prints `x+1` for every number `x` in the list. (Hint: you will need to start at the first element in `mylist`, which you learned how to retrieve in last week's lab, and then retrieve each item in the list in order. You'll want the condition to evaluate to `False` when you reach the end of the list.)

Solution 5.

```
mylist = [1,2,3,4,5]
index = 0
#don't want to go past end of list
while index < len(mylist):
    #retrieve element, print x + 1
    x = mylist[index]
    print x + 1
    #now increase index to move along the list
    index = index + 1
```

Checkpoint 1. You have reached a checkpoint! Please call the PGTA or lecturer over to check what you've done so far.

4 For Loops, Revisited

Even though we saw the uses of `for` loops on their own last week, they're often most useful when paired with `if` statements inside the body, as we do not usually want to execute the exact same code on every element in a sequence, but rather choose what to execute based on its value.

In addition to their usefulness in terms of functionality, integrating `if` statements into `for` loops is also beneficial in terms of efficiency. Consider the following situation: we are looking through a list of strings `mylist` to try to find the entry that contains `'secu2002'` as a substring (let's suppose we know there is only one such entry). If we were doing this naïvely, we would write:

[1] If you execute this code, either in the interactive interpreter or by including it in a file that you run, you must manually break the execution using Control-C. You've been warned!

```
whole_string = ‘’
for x in mylist:
    if ‘secu2002’ in x:
        whole_string = x
```

At the end of this loop, the variable `whole_string` would have the value of the entry that we want. In the process, however, we would have iterated through the entire list, which might have millions of entries. To save ourselves the time, we can use the `break` statement, which in terms of control flow acts to stops the iteration and continues to whatever code is after the loop. So, if we instead used

```
whole_string = ‘’
for x in mylist:
    if ‘secu2002’ in x:
        whole_string = x
        break
```

then we would iterate only up until we found the entry we were looking for. The `continue` statement has somewhat similar behavior, but rather than stop the iteration entirely, it skips the remainder of the body of the loop and moves on to the next entry in the list. Again, both of these control flow statements are most useful in providing efficiency, rather than functionality.

5 Input and Output

As we saw in the first lab when we tried to run our first program and saw nothing, retrieving inputs and producing outputs (also shortened to input/output, or i/o for short) is one of the most crucial components of programming. After all, programs do not exist in isolation, and need to interact meaningfully with the outside world (by, e.g., reading in data or printing out the results of a computation) in order to be truly useful. We've already seen one form of input/output, which is the `print` function: this is responsible for communicating the value of some series of expressions to the world outside of the program. We have not, however, seen any ways for programs to take in outside information, aside from us typing it into the program ourselves.

5.1 File input

Let's start with the simple example of opening a file, which syntactically can be achieved using

```
f = open(filename, ‘r’)
```

The keyword `open` tells Python that we want to open a file, the filename `filename` tells it which file we want to open, and `‘r’` tells it we want to read the contents of the file (the other options are `‘w’`, to overwrite the contents of the file, and `‘a’`, to append to the contents of the file). This is the default value of this flag, meaning we could also run just `f = open(filename)` to read the file, and Python looks for `filename` in the current working directory. If the file is in a different directory, you'll need to specify this in `filename` (see Lab 01 for a reminder of the structure of file systems).

If you don't assign the value of `open`, you might be a bit disappointed by what you see. For a file `1_to_5.txt` that has the following contents

```
1
2
3
4
5
```

running `open` directly looks like this:

```
>>> open(‘1_to_5.txt’,‘r’)
<open file ‘1_to_5.txt’, mode ‘r’ at 0x10a61a5d0>
```

Indeed, `open` doesn't return the file contents in a nicely printed format, but instead returns a file opener. To get the actual text contents of the file, we can run `f.read()`, although again you might be disappointed by the outcome:

```
>>> f.read()
'1\n2\n3\n4\n5\n'
```

These are in fact the contents of the file, but it's not very human-readable. You also have to be careful to assign the contents immediately after the file is read; otherwise, you again won't like what you see:

```
>>> f.read()
'1\n2\n3\n4\n5\n'
>>> f.read()
''
```

To get a nicer list, we could run `lines = f.readlines()` (again, this function will not return the same thing the second time it is run, so be sure to assign its value to some variable). Even better, we could iterate over the file opener directly, as in:

```
>>> for line in f:
...     print line
...
1
2
3
4
5
```

When you're done with a file, you always want to *close* it by running `f.close()`.

Some types of files may have a specific format, like a CSV file (CSV is short for *comma-separated values*). We will learn specific ways to deal with specific file types later on, but for now we can use the `split` function for strings to deal with CSV, which is essentially the opposite of the `join` function we saw last week. (In fact, for any string `x` and separator string `sep`, it is the case that `sep.join(x.split(sep)) = x`.) This function is run by a string, so can be called using `mystr.split()`. The default argument is to split a sentence into its individual words; i.e., to split the string into sequences of characters separated by spaces. For example:

```
>>> 'secu2002 is great'.split()
['secu2002', 'is', 'great']
```

If you supply the function with an optional argument, like `','`, `'\t'`, or `'\n'`, then it will split the string using this separator instead. The first of these can be used to split each line in a CSV file into its individual fields, the second to split tab-separated values, and the third to split a spreadsheet into individual rows (which can in turn be split into their columns using a comma).

Task 6. In the `data` directory of the master Git repository, you can find a file called `hello_world.txt`. In a file `read_hello.py`, write code to open this file and, for every line in it, print out the first four characters.

Solution 6. The contents of `read_hello.py` are as follows (but the filename will differ based on how your file system is structured):

```
f = open('../../../data/hello_world.txt','r')
for line in f:
    print line[:4]
```

5.2 File output

Writing to a file is quite similar to reading from one. To start, you use the same `open` function, but provide the flag indicating you want to write to the file, as in `f = open('1_to_5.txt','w')`. Be very careful when running this function though: using `'w'` will immediately erase the contents of the `'1_to_5.txt'` file, if it exists. In this sense, using the `'a'` flag instead is much safer.

To write to the file, you run something like `f.write('Here is some content.')`. When you're done writing to the file, you run `f.close()`.

Task 7. Create a file `write_world.py`. Within this file, write the code to create a file `hello_world.txt` and fill it with the contents "Hello, world!"

Solution 7. The contents of `write_hello.py` are as follows:

```
f = open('hello_world.txt','w')
f.write('Hello, world!')
f.close()
```

5.3 User-defined input

One additional way to incorporate outside input is to allow a user to directly type their own text strings. This is achieved using the `raw_input` function, which takes as input a string to print out to the user when requesting their input, and produces as output whatever the user entered. (In Python 3, this is now just `input`.) Syntactically, this looks as follows:

```
>>> x = raw_input('What is your name? ')
What is your name? Sarah
>>> x
'Sarah'
```

If instead you have the `raw_input` line in a file, then when you run the file you will see the interaction in the command line or console.

The `raw_input` function always returns a string, but it is possible to cast it as another value; for example, if we're collecting ages, then we might want an `int` instead. It is possible to do this by just running `int(x)`, but it is also possible to do it using the more powerful `eval` function. This function takes a string as input and treats it as a Python expression, meaning it evaluates whatever is contained inside the string. For example:

```
>>> x = 4
>>> eval('x + 1')
5
```

Given that `eval` then treats strings as executable code, you should be extremely careful when running it on user-defined inputs! This can lead to dangerous security vulnerabilities, and it is much safer to use things like `int(x)` instead. In fact, any time you use `eval` it is a good idea to ask yourself if there is something else you could be using instead, as usually there is a much safer and better way to do things.

6 Putting It All Together

Now that we have conditional statements, loops, and i/o, we have almost everything we need to start building real programs. To demonstrate this, we'll be implementing today the game Hangman.[2]

For our purposes, Hangman proceeds as follows: a secret phrase is chosen, and the player is given that phrase with all letters replaced by `_` (but with spaces and other special characters preserved). In every round, the player is allowed to guess a single letter. If they have guessed a letter in the phrase, they are given the hidden phrase with that letter filled in, along with all the letters they have guessed so far. For example, if we were playing Hangman with the word 'sarah', it could proceed as follows:

- Round 0: The player is given '____'. They guess 'a'.
- Round 1: The player is given '_a_a_'. They guess 'x'.
- Round 2: The player is given '_a_a_' (since 'x' wasn't in the word). They guess 'r'.
- Round 3: The player is given '_ara_'. They guess 's'.

[2]If you're completely unfamiliar with the rules of this game, you may want to read about it here: `en.wikipedia.org/wiki/Hangman_(game)`.

- Round 4: The player is given 'sara_'. They guess 'h'.
- Round 5: The player is given 'sarah' and, now that they have guessed the word, the game is over.

Task 8. To start, create a file `secret_phrase.txt` containing, on a single line and all in lowercase letters, the word or phrase you want people to guess.

Next, create a file `hangman.py`, and load the secret phrase into a variable `secret_phrase` using file input.

Create another variable `shown_phrase` that is the initial phrase to show to the player; i.e., a phrase containing a `_` (or other) character where every letter should be (but keeping things like spaces, hyphens, and apostrophes the same). There are many ways you could do this, but please restrict yourselves to only things we've covered already in labs, and remember that `for` loops and `if` statements are your friends.

Now that you have everything set up, you're ready to begin interacting with the player. As in the example above, you want to continue playing until they have guessed the entire phrase. In each round, the current value of the phrase should be shown to the user, who should then be allowed to guess one letter. Your job is to update the value of `shown_phrase` to incorporate the letter they have guessed. This process is not too different to how you set up `shown_phrase` in the first place, but should be run only if the letter is in the phrase and only if they haven't guessed the letter already. At the end, when the player has guessed the phrase, you should print out how many tries it took them to do it.

To make your lives easier, it's okay to assume that the player is playing correctly, meaning they always enter a single lowercase letter. To handle as many edge cases as possible though, you can earn one bonus point if you include code to ask the user again for input every time they enter something else (like a number or multiple letters).

Solution 8. See `hangman.py`.

Checkpoint 2. You have reached a checkpoint! Please call the PGTA or lecturer over to check what you've done so far.

7 Handing In

As always, feel free to play around some more with all of the concepts we've covered in this lab. Once you are done, add all `.py` and `.txt` files into a `lab03` directory; this should mean adding:

- `swap.py`
- `while_list.py`
- `read_hello.py`
- `write_hello.py`
- `hangman.py`
- `secret_phrase.txt`

Push these files, along with the usual `partner.txt`, to the remote repository, and you're done! Please check with one of us on your way out.