A picture containing drawing

Description automatically generated

**FAQ:**

1. Introduction to Python control flow statements

**1. Conditional statements**

**Indentation**

Some other languages use braces to show where blocks of code begin and end. In Python we use indentation to enclose blocks of code. For example, if statements use indentation to tell Python what code is inside and outside of different clauses.

In Python, indents conventionally come in multiples of four spaces. Be strict about following this convention, because changing the indentation can completely change the meaning of the code. If you are working on a team of Python programmers, it's important that everyone follows the same indentation convention!

#### **Spaces or Tabs?**

The Python Style Guide recommends using 4 spaces to indent, rather than using a tab. Whichever you use, be aware that "Python 3 disallows mixing the use of tabs and spaces for indentation."

**2.Boolean expression for conditional statements**

If statements sometimes use more complicated boolean expressions for their conditions. They may contain multiple comparisons operators, logical operators, and even calculations. Examples:

**if** 18.5 <= weight / height\*\*2 < 25:

print("BMI is considered 'normal'")

**if** is\_raining **and** is\_sunny:

print("Is there a rainbow?")

**if** (**not** unsubscribed) **and** (location == "USA" **or** location == "CAN"):

print("send email")

For complicated conditions you might need to combine some ands, ors and nots together. Use parentheses if you need to make the combinations clear.

However simple or complex, the condition in an if statement must be a Boolean expression that evaluates to either True or False and it is this value that decides whether the indented block in an if statement executes or not.

# Good and Bad Examples

Here are some things to keep in mind while writing Boolean expressions for your if statements.

### 1. Don't use True or False as conditions

*# Bad example*

**if** **True**:

print("This indented code will always get run.")

While "True" is a valid Boolean expression, it's not useful as a condition since it always evaluates to True, so the indented code will always get run. Similarly, if False is not a condition you should use either - the statement following this if statement would never be executed.

*# Another bad example*

**if** is\_cold **or** **not** is\_cold:

print("This indented code will always get run.")

Similarly, it is useless to use any condition that you know will always evaluate to True, like this example above. A Boolean variable can only be True or False, so either is\_cold or not is\_cold is always True, and the indented code will always be run.

### 2. Be careful writing expressions that use logical operators

Logical operators and, or and not have specific meanings that aren't quite the same as their meanings in plain English. Make sure your Boolean expressions are being evaluated the way you expect them to.

*# Bad example*

**if** weather == "snow" **or** "rain":

print("Wear boots!")

This code is valid in Python, but it is not a boolean expression, although it reads like one. The reason is that the expression to the right of the or operator, "rain", is not a boolean expression - it's a string! Later we'll discuss what happens when you use non-boolean-type objects in place of booleans.

### 3. Don't compare a boolean variable with == True or == False

This comparison isn’t necessary, since the boolean variable itself is a boolean expression.

*# Bad example*

**if** is\_cold == **True**:

print("The weather is cold!")

This is a valid condition, but we can make the code more readable by using the variable itself as the condition instead, as below.

*# Good example*

**if** is\_cold:

print("The weather is cold!")

# Truth Value Testing

If we use a non-boolean object as a condition in an if statement in place of the boolean expression, Python will check for its truth value and use that to decide whether or not to run the indented code. By default, the truth value of an object in Python is considered True unless specified as False in the documentation.

Here are most of the built-in objects that are considered False in Python:

* constants defined to be false: None and False
* zero of any numeric type: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)
* empty sequences and collections: '"", (), [], {}, set(), range(0)

Example:

errors = 3

**if** errors:

print("You have {} errors to fix!".format(errors))

**else**:

print("No errors to fix!")

In this code, errors have the truth value True because it's a non-zero number, so the error message is printed. This is a nice, succinct way of writing an if statement.

**3. For loops**

Python has two kinds of loops - for loops and while loops. A for loop is used to "iterate", or do something repeatedly, over an **iterable**.

An **iterable** is an object that can return one of its elements at a time. This can include sequence types, such as strings, lists, and tuples, as well as non-sequence types, such as dictionaries and files.

### Example

Let's break down the components of a for loop, using this example with the list cities:

cities = ['new york city', 'mountain view', 'chicago', 'los angeles']

**for** city **in** cities:

print(city)

print("Done!")

#### Components of a for Loop

1. The first line of the loop starts with the for keyword, which signals that this is a for loop
2. Following that is city in cities, indicating city is the iteration variable, and cities is the iterable being looped over. In the first iteration of the loop, city gets the value of the first element in cities, which is “new york city”.
3. The for loop heading line always ends with a colon :
4. Following the for loop heading is an indented block of code, the body of the loop, to be executed in each iteration of this loop. There is only one line in the body of this loop - print(city).
5. After the body of the loop has executed, we don't move on to the next line yet; we go back to the for heading line, where the iteration variable takes the value of the next element of the iterable. In the second iteration of the loop above, city takes the value of the next element in cities, which is "mountain view".
6. This process repeats until the loop has iterated through all the elements of the iterable. Then, we move on to the line that follows the body of the loop - in this case, print("Done!"). We can tell what the next line after the body of the loop is because it is unindented. Here is another reason why paying attention to your indentation is very important in Python!

Executing the code in the example above produces this output:

new york city

mountain view

chicago

los angeles

Done!

You can name iteration variables however you like. A common pattern is to give the iteration variable and iterable the same names, except the singular and plural versions respectively (e.g., 'city' and 'cities').

**4. While loops**

For loops are an example of "definite iteration" meaning that the loop's body is run a predefined number of times. This differs from "indefinite iteration" which is when a loop repeats an unknown number of times and ends when some condition is met, which is what happens in a while loop. Here's an example of a while loop.

card\_deck = [4, 11, 8, 5, 13, 2, 8, 10]

hand = []

*# adds the last element of the card\_deck list to the hand list*

*# until the values in hand add up to 17 or more*

**while** sum(hand) < 17:

hand.append(card\_deck.pop())

This example features two new functions. sum returns the sum of the elements in a list, and pop is a list method that removes the last element from a list and returns it.

### Components of a While Loop

1. The first line starts with the while keyword, indicating this is a while loop.
2. Following that is a condition to be checked. In this example, that's sum(hand) <= 17.
3. The while loop heading always ends with a colon :.
4. Indented after this heading is the body of the while loop. If the condition for the while loop is true, the code lines in the loop's body will be executed.
5. We then go back to the while heading line, and the condition is evaluated again. This process of checking the condition and then executing the loop repeats until the condition becomes false.
6. When the condition becomes false, we move on to the line following the body of the loop, which will be unindented.

The indented body of the loop should modify at least one variable in the test condition. If the value of the test condition never changes, the result is an infinite loop!

**5. For loops VS While loops**

Now that you are familiar with both for and while loops, let's consider when it's most helpful to use each of them.

**for** loops are ideal when the **number of iterations is known or finite**.

Examples:

* When you have an iterable collection (list, string, set, tuple, dictionary)
  + for name in names:
* When you want to iterate through a loop for a definite number of times, using range()
  + for i in range(5):

**while** loops are ideal when the **iterations need to continue until a condition is met**.

Examples:

* When you want to use comparison operators
  + while count <= 100:
* When you want to loop based on receiving specific user input.
  + while user\_input == 'y':

Check out these webpages to see more examples on this topic:

* StackOverflow [discussion](https://stackoverflow.com/questions/920645/when-to-use-while-or-for-in-python)
* Wiki page on [Python.org](https://wiki.python.org/moin/WhileLoop)

**6. Zip and Enumerate**

### Zip

zip returns an iterator that combines multiple iterables into one sequence of tuples. Each tuple contains the elements in that position from all the iterables. For example, printing

list(zip(['a', 'b', 'c'], [1, 2, 3])) would output [('a', 1), ('b', 2), ('c', 3)].

Like we did for range() we need to convert it to a list or iterate through it with a loop to see the elements.

You could unpack each tuple in a for loop like this.

letters = ['a', 'b', 'c']

nums = [1, 2, 3]

**for** letter, num **in** zip(letters, nums):

print("{}: {}".format(letter, num))

In addition to zipping two lists together, you can also unzip a list into tuples using an asterisk.

some\_list = [('a', 1), ('b', 2), ('c', 3)]

letters, nums = zip(\*some\_list)

This would create the same letters and nums tuples we saw earlier.

### Enumerate

enumerate is a built in function that returns an iterator of tuples containing indices and values of a list. You'll often use this when you want the index along with each element of an iterable in a loop.

letters = ['a', 'b', 'c', 'd', 'e']

**for** i, letter **in** enumerate(letters):

print(i, letter)

This code would output:

0 a

1 b

2 c

3 d

4 e

**---X---**