AN7914 Week 02 Python

February 7, 2024

1 Week 2 Python

1.1 Conditionals

In this week's session, we're diving into conditionals and loops in Python. Conditionals are a fundamental concept in programming, allowing your code to make decisions based on certain criteria. This enables your programs to respond differently to different inputs or situations, making your code more dynamic and versatile.

But first we look at Comparison Operators.

1.1.1 Understanding Comparison Operators in Python

Python provides several "operators" for making mathematical comparisons, which are essential for controlling the flow of your programs:

- `>` (Greater Than) and `<` (Less Than): Used to compare two values to determine if one is larger or smaller than the other.
- `>=` (Greater Than or Equal To): Checks if a value is greater than or equal to another.
- `<=` (Less Than or Equal To): Checks if a value is less than or equal to another.
- `==` (Equal To): Used for equality checks between two values. Remember, double equals (`==`) are for comparison, whereas a single equal sign (`=`) is used for assigning values.
- `!=` (Not Equal To): Determines if two values are not the same.

These operators allow you to ask questions about the relationship between values and variables, enabling decision-making in your code based on those comparisons.

1.1.2 What are if Statements?

The if statement is the simplest form of a conditional in Python. It checks a condition: if the condition is True, it executes a block of code. Understanding if statements is crucial for writing conditional logic in your programs. Let's start with a basic example to see how if statements work

1.1.3 if statements

[1]:
```python
# Example of a simple if statement
x = int(input('Enter a number: '))  # Prompt the user to enter a number

# Check if the number is positive
if x > 0:
    print('x is positive')  # This line executes only if x is greater than 0
```

```
Enter a number: 2
x is positive
```

If statements use `bool` or boolean values (true or false) to decide whether or not to execute. If the statement of `x > y` is true, the compiler will register it as `true` and execute the code.

[2]:
```python
# Prompt the user to input a value for x and convert it to an integer
x = int(input('What is x?'))

# Prompt the user to input a value for y and convert it to an integer
y = int(input('What is y?'))

# Check if x is less than y
if x < y:
    # If true, print that x is less than y
    print('x is less than y')

# Check if x is greater than y
if x > y:
    # If true, print that x is greater than y
    print('x is greater than y')
```

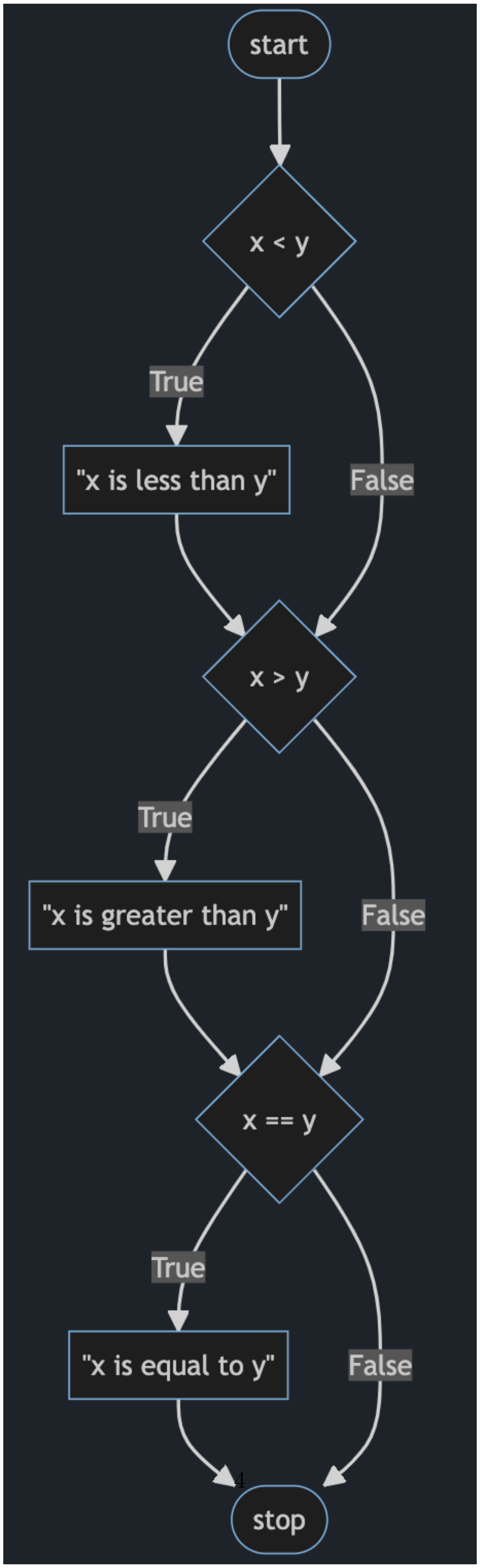
```
What is x?2
What is y?2
```

We can revise our code as follows

[3]:
```python
# Prompt the user to input a value for x and convert it to an integer
x = int(input('What is x?'))

# Prompt the user to input a value for y and convert it to an integer
y = int(input('What is y?'))

# Check if x is less than y
if x < y:
    # If the condition is true, print that x is less than y
    print('x is less than y')

# Check if x is greater than y
if x > y:
    # If the condition is true, print that x is greater than y
    print('x is greater than y')

# Check if x is equal to y
if x == y:
    # If the condition is true, print that x equals y
    print('x equals y')
```

```
What is x?2
What is y?2
x equals y
```

Notice how you are providing a series of `if` statements. First, the first `if` statement is evaluated. Then, the second `if` statement runs its evaluation. Finally, the last `if` statement runs its evaluation. This flow of decisions is called "control flow." See the image below

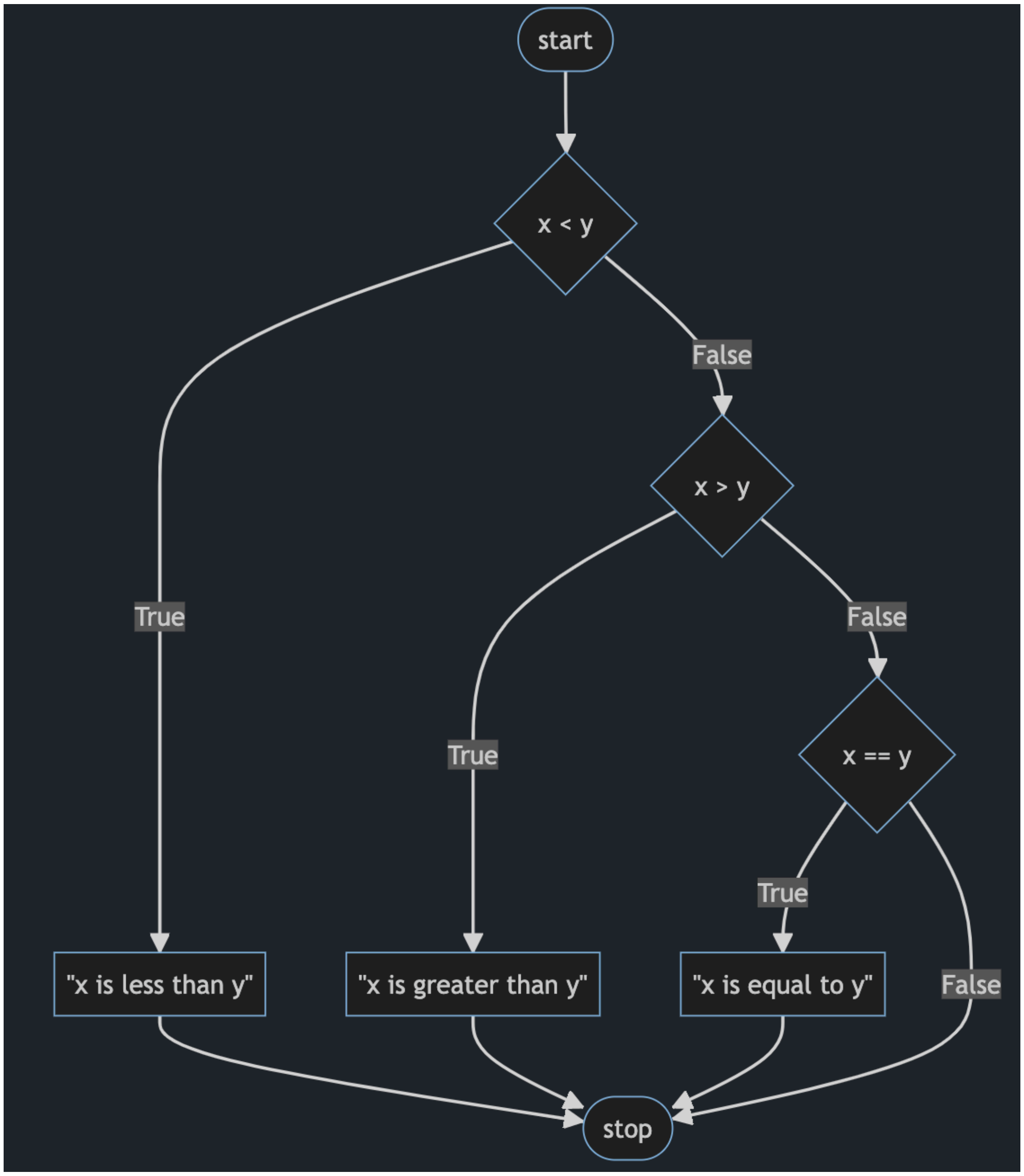

start
x < y
True
"x is less than y"
False
x > y
True
"x is greater than y"
False
x == y
True
"x is equal to y"
False
stop

This program can be improved by not asking three consecutive questions. After all, not all three questions can have an outcome of true! We can improve our program as follows:

[4]:
```python
# Prompt the user to enter a value for x and convert it to an integer
x = int(input("What's x? "))

# Prompt the user to enter a value for y and convert it to an integer
y = int(input("What's y? "))

# Compare x and y to determine their relationship
if x < y:
    # If x is less than y, print that x is less than y
    print("x is less than y")
elif x > y:
    # If x is greater than y, print that x is greater than y
    print("x is greater than y")
elif x == y:
    # If x is equal to y, print that x is equal to y
    print("x is equal to y")
```

```
What's x? 2
What's y? 2
x is equal to y
```

Notice how the use of `elif` allows the program to make less decisions. First, the `if` statement is evaluated. If this statement is found to be `true`, all the `elif` statements not be run at all. However, if the `if` statement is evaluated and found to be `false`, the first `elif` will be evaluated. If this is `true`, it will not run the final evaluation.

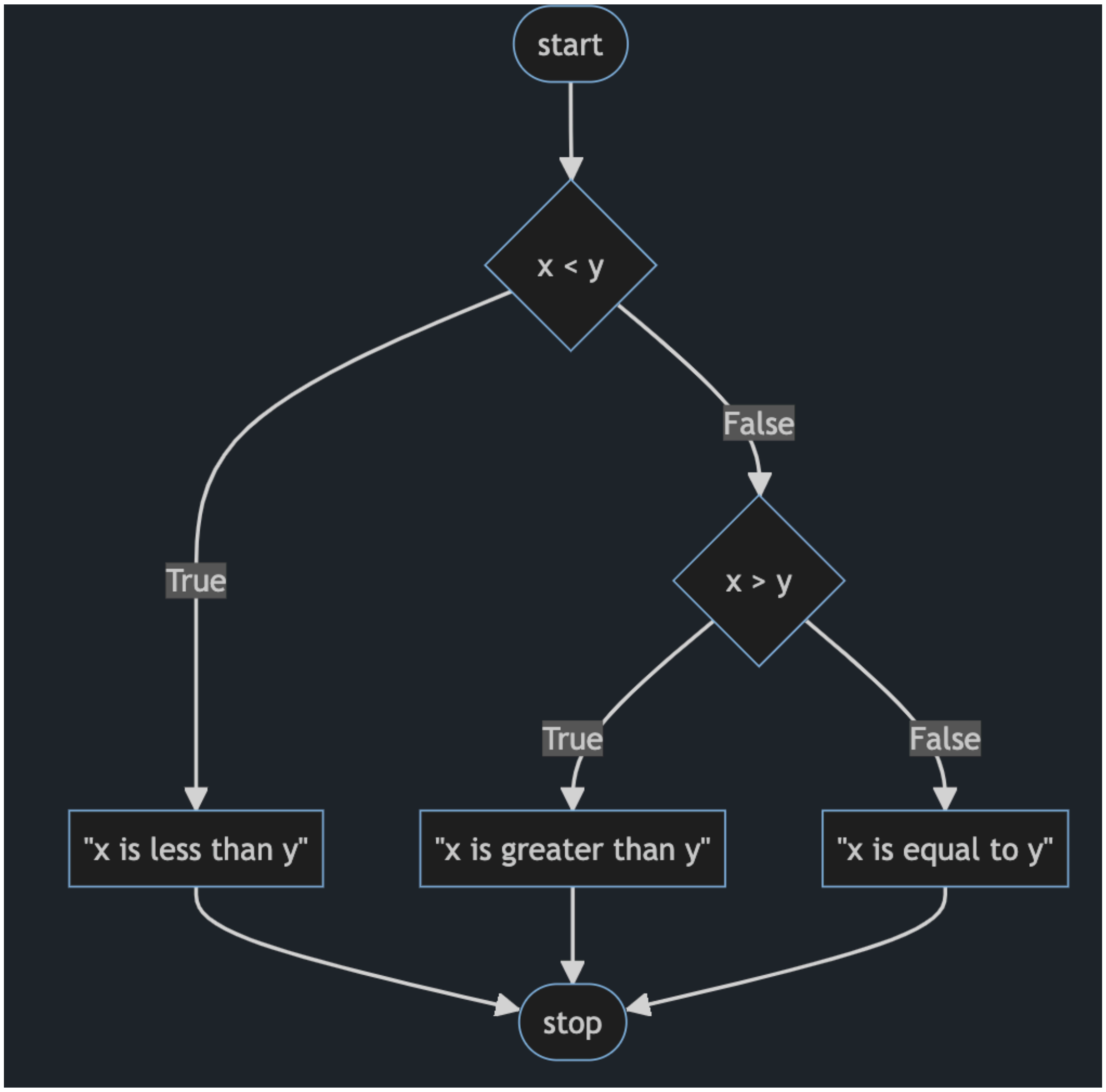


There is one final improvement we can make to our program. Notice how logically `elif x == y` is not a necessary evaluation to run. After all, if logically `x` is not less than `y` AND `x` is not greater than `y`, `x` MUST equal `y`. Therefore, we don't have to run `elif x == y`. We can create a "catch-all," default outcome using an `else` statement. We can improve as follows:

[5]:
```python
# Prompt the user to input a value for x and convert it to an integer
x = int(input('What is x?'))

# Prompt the user to input a value for y and convert it to an integer
y = int(input('What is y?'))
```

```python
# Compare x and y to determine their relationship
if x < y:
    # If x is less than y, inform the user that x is less than y
    print('x is less than y')
elif x > y:
    # If x is greater than y, inform the user that x is greater than y
    print('x is greater than y')
else:
    # If neither of the above conditions are true, x must equal y
    print('x equals y')
```

```
What is x?2
What is y?2
x equals y
```



1.1.4 or

`or` allows your program to decide between one or more alternatives. For example, we could further edit our program as follows:

[6]:
```python
# Prompt the user to input a value for x and convert it to an integer
x = int(input('What is x?'))

# Prompt the user to input a value for y and convert it to an integer
y = int(input('What is y?'))

# Check if x is not equal to y using a logical OR operator
if x < y or x > y:
    # If x is less than y or x is greater than y, it means x is not equal to y
    print('x is not equal to y')
else:
    # If neither condition is true, then x must be equal to y
    print('x equals y')
```

```
What is x?2
What is y?2
x equals y
```

Now we can also use `!=` operator. This helps us ask a simple question 'is x not equal to y?'

[7]:
```python
# Prompt the user to input a value for x and convert it to an integer
x = int(input('What is x?'))

# Prompt the user to input a value for y and convert it to an integer
y = int(input('What is y?'))

# Check if x is not equal to y
if x != y:
    # If x is not equal to y, print that x is not equal to y
    print('x is not equal to y')
else:
    # If x is equal to y, print that x equals y
    print('x equals y')
```

```
What is x?2
What is y?2
x equals y
```

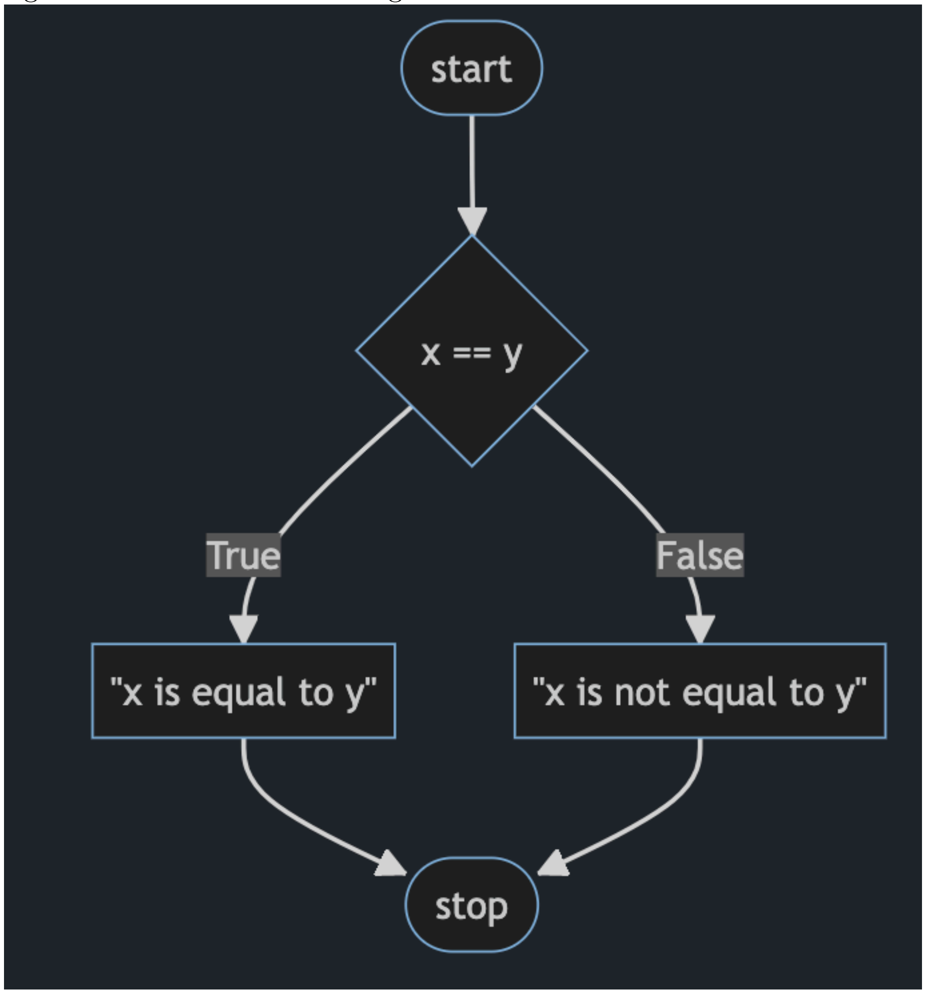
We could also do something like the following. Now we are using `==` operator.

[8]:
```python
# Prompt the user to input a value for x and convert that input to an integer
x = int(input('What is x?'))

# Prompt the user to input a value for y and convert that input to an integer
y = int(input('What is y?'))
```

```python
# Check if the values of x and y are equal
if x == y:
    # If x is equal to y, print that x equals y
    print('x equals to y')
else:
    # If x is not equal to y, print that x is not equal to y
    print('x is not equal to y')
```

```
What is x?2
What is y?2
x equals to y
```

Notice that the `==` operator evaluates if what is on the left and right are equal to one another. That use of double equal signs is very important. If you use only one equal sign, an error will likely be thrown by the compiler. One equal sign is for assignment. See the following flow chart to understand how double equal operator works.



1.1.5 and

Similar to `or`, `and` can be used within conditional statements.

[9]:
```python
# Prompt the user to input their score and convert that input to an integer
score = int(input("Give me your score: "))

# Check if the score is between 90 and 100, inclusive
if score >= 90 and score <= 100:
    # If the score is in this range, print 'Grade A'
    print('Grade A')
# Check if the score is between 80 and 89, inclusive
elif score >= 80 and score < 90:
    # If the score is in this range, print 'Grade B'
    print('Grade B')
# Check if the score is between 70 and 79, inclusive
elif score >= 70 and score < 80:
    # If the score is in this range, print 'Grade C'
    print('Grade C')
# Check if the score is between 60 and 69, inclusive
elif score >= 60 and score < 70:
    # If the score is in this range, print 'Grade D'
    print('Grade D')
# If the score does not fit any of the above categories, it is below 60
else:
    # Print 'Fail' for scores below 60
    print('Fail')
```

```
Give me your score: 90
Grade A
```

Further improving the code below.

[10]:
```python
# Prompt the user to input their score and convert that input to an integer
score = int(input("Give me your score: "))

# Check if the score falls within the range for Grade A (90 to 100, inclusive)
if 90 <= score <= 100:
    # If true, print 'Grade A'
    print('Grade A')
# Check if the score falls within the range for Grade B (80 to 89, inclusive)
elif 80 <= score < 90:
    # If true, print 'Grade B'
    print('Grade B')
# Check if the score falls within the range for Grade C (70 to 79, inclusive)
elif 70 <= score < 80:
    # If true, print 'Grade C'
    print('Grade C')
# Check if the score falls within the range for Grade D (60 to 69, inclusive)
elif 60 <= score < 70:
    # If true, print 'Grade D'
    print('Grade D')
```

```
# If the score does not meet any of the above conditions, it is below 60
else:
    # Print 'Fail' for scores below 60
    print('Fail')
```

```
Give me your score: 91
Grade A
```

[11]:
```
# Prompt the user to input their score and convert that input to an integer
score = int(input("Give me your score: "))

# Check if the score is 90 or above
if score >= 90:
    # If true, the grade is A
    print('Grade A')
# If the score wasn't high enough for Grade A, check if it's 80 or above
elif score >= 80:
    # If true, the grade is B
    print('Grade B')
# If the score wasn't high enough for Grade B, check if it's 70 or above
elif score >= 70:
    # If true, the grade is C
    print('Grade C')
# If the score wasn't high enough for Grade C, check if it's 60 or above
elif score >= 60:
    # If true, the grade is D
    print('Grade D')
# If the score wasn't high enough for Grade D, it is below 60
else:
    # The grade is F, indicating failure
    print('Fail')
```

```
Give me your score: 90
Grade A
```

1.2 Loops/Iteration

Loops are powerful constructs in Python that allow you to repeat a block of code multiple times. Python offers two types of loops: `for` loops, which iterate over a sequence (like a list or a range), and `while` loops, which continue as long as a condition remains true.

Essentially, loops are a way to do something over and over again.

[12]:
```
print('meow')
print('meow')
print('meow')
```

```
meow
meow
```

```
meow
```

Running this code you'll notice that the program meows three times. As a side you could also do it like this below

[13]:
```
print(3*' meouw')
```

```
 meouw meouw meouw
```

If you to print them on different lines you could try this:

[14]:
```
print(3*' meouw \n')
```

```
 meouw
 meouw
 meouw
```

The code `print(3 * ' meouw \n')` prints the word "meouw" three times on separate lines. This works because:

- `3 * ' meouw \n'` repeats the string " meouw " followed by a newline character `\n` three times.
- The newline character `\n` tells Python to start a new line, so each "meouw" appears on its own line.

In developing as a programmer, you want to consider how one could improve areas of one's code where one types the same thing over and over again. Imagine where one might want to "meow" 500 times. Would it be logical to type that same expression of `print("meow")` over and over again?

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. Because iteration is so common, Python provides several language features to make it easier. One form of iteration in Python is the `while` statement.

[15]:
```
# Initialize the variable i with a value of 3
i = 3

# Start a while loop that continues as long as i is not equal to 0
while i != 0:
    # Print 'meouw' each time the loop runs
    print('meouw')
    # Decrease the value of i by 1 on each iteration
    i = i - 1
```

```
meouw
meouw
meouw
```

More formally, here is the flow of execution for a while statement: 1. Evaluate the condition, yielding `True` or `False`. 2. If the condition is `False`, exit the `while` statement and continue execution at the next statement. 3. If the condition is `True`, execute the body and then go back to step 1. This type of flow is called a loop because the third step loops back around to the top. We call each time we execute the body of the loop an iteration. Here is another example of a `while` loop.

[16]:
```python
# Initialize the variable n with a value of 5
n = 5

# Start a while loop that continues as long as n is greater than 0
while n > 0:
    # Print the current value of n
    print(n)
    # Decrease the value of n by 1
    n = n - 1
    # After decrementing n, print 'Blastoff!'
    print('Blastoff!')
```

```
5
Blastoff!
4
Blastoff!
3
Blastoff!
2
Blastoff!
1
Blastoff!
```

You can almost read the `while` statement as if it were English. It means, "While `n` is greater than `0`, display the value of `n` and then reduce the value of `n` by `1`. When you get to `0`, exit the `while` statement and display the word `Blastoff`!".

For the above loop, we would say, "It had five iterations", which means that the body of the loop was executed five times.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. We call the variable that changes each time the loop executes and controls when the loop finishes the iteration variable. If there is no *iteration variable*, the loop will repeat forever, resulting in an *infinite loop*.

1.2.1 for loop

A `for` loop is a different type of loop. To best understand a `for` loop, it's best to begin by talking about a new variable type called a `list` in Python. As in other areas of our lives, we can have a grocery list, a to-do list, etc. A `for` loop iterates through a list of items. For example, we can modify our code for 'meouw' as follows.

[17]:
```python
# Iterate over a list containing 0, 1, and 2
for i in [0, 1, 2]:
    # For each iteration, print 'meouw'
    print('meouw')
```

```
meouw
meouw
meouw
```

Notice how clean this code is compared to your previous `while` loop code. In this code, `i` begins with `0`, meows, `i` is assigned `1`, meows, and, finally, `i` is assigned `2`, meows, and then ends.

[18]:
```python
# Iterate over a list containing the numbers 2, 100, and 21
for i in [2, 100, 21]:
    # For each iteration, regardless of the value of i, print 'meouw'
    print('meouw')
```

```
meouw
meouw
meouw
```

While this code accomplishes what we want, there are some possibilities for improving our code for extreme cases. At first glance, our code looks great. However, what if you wanted to iterate up to a million? It's best to create code that can work with such extreme cases. Accordingly, we can improve our code as follows:

[19]:
```python
# Iterate over a range of 3 numbers starting from 0 up to, but not including, 3
for i in range(3):
    # For each iteration, print 'meouw'
    print('meouw')
```

```
meouw
meouw
meouw
```

Notice how `range(3)` provides back three values (0, 1, and 2) automatically. This code will execute and produce the intended effect, meowing three times.

[ ]:
```python

```