Python Class Notes

Clarusway 

Control Flow Statements

Nice to have VSCode Extentions:

- Jupyter

Needs

- Python (for Windows OS: add to the path while installing!)

Summary

- Introduction
- Conditional Statements
  - If
  - Match (NEW! > Python 3.10) (Optional)
- Looping Statements
  - While
  - For
  - The built-in range() function
  - Nested for loops
  - While vs For
- Transfer Statements
  - break
  - continue
  - return

- The pass Statement

Introduction

Control flow statements in Python are statements that allow you to control the execution of your code based on certain conditions. They let you alter the flow of the program depending on whether a particular condition is met or not. In Python, there are three main types of control flow statements:

- **Conditional Statements**: These statements allow you to execute a particular block of code only if a certain condition is met. The most commonly used conditional statement in Python is the `if statement`. You can also use the elif and else statements to create more complex conditions. The `match statement` was introduced in Python 3.10 as a new feature.
- **Looping Statements**: These statements allow you to execute a block of code repeatedly until a certain condition is met. The most commonly used looping statement in Python is the `for loop`. You can also use the `while loop` to create more complex loops.
- **Transfer Statements**: These statements allow you to transfer control to a different part of your code. The most commonly used transfer statement in Python is the `break statement`, which allows you to exit a loop early. You can also use the `continue statement` to skip over certain iterations of a loop, and the `return statement` to exit a function and return a value.

These control flow statements give you a lot of flexibility in how you structure your code and how it executes. They allow you to create more complex programs that can adapt to changing conditions and user input.

Conditional Statements

If

The if statement is a conditional statement in Python that allows you to execute a block of code only if a certain condition is true. Here's the basic syntax of an if statement:

```
if condition:
    # Code to be executed if condition is true
```

In this syntax, condition is any expression that evaluates to a Boolean value (True or False). If the condition is true, the code block following the if statement is executed. If the condition is false, the code block is skipped, and the program moves on to the next statement.

Here's an example of an if statement in action:

```
my_num = 5
if my_num > 0:
    print(f"{my_num} is positive")
```

You can also use the elif and else statements to create more complex conditions. Here's an example:

```
x = 0
if x > 0:
    print("x is positive")
elif x == 0:
    print("x is zero")
else:
    print("x is negative")
```

More examples:

```
# Check if the list is empty or not:
my_list = [1]
if my_list:
    print("The list is not empty!")

# Structure:
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")

# You can also have an else without the elif
```

- TIP: While using if, put the most specific condition to the top.

Match

The match statement provides a way to perform pattern matching on values, making it easier to handle complex data structures and control flow. It is similar to the switch statement in other programming languages, but with a more powerful and flexible syntax.

Here is an example of how the match statement can be used:

```
fruit = 'avacado'

match fruit:
    case "apple":
        print("This is an apple.")
    case "banana":
        print("This is a banana.")
    case "orange":
```

```
        print("This is an orange.")
    case _:
        print("I don't know what this fruit is.")
```

Note the last block: the variable name, _, acts as a wildcard and insures the subject will always match. The use of _ is optional.

In this example, the match statement is used to match the fruit argument against different cases. If the fruit argument matches one of the cases, the corresponding code block is executed. If none of the cases match, the default case _ is executed, which prints a message indicating that the fruit is unknown.

Here are some examples:

```
status = 400

match status:
    case 400:
        print("Bad request")
    case 404:
        print("Not found")
    case 418:
        print("I'm a teapot")
    case _:
        print("Something's wrong with the internet")
```

You can combine several literals in a single pattern using | (or):

```
status = 401

match status:
    case 401 | 403 | 404:
        print("Not allowed")
```

This is a new feature for Python. See the documentation for the details.

Looping Statements

While

With the while loop we can execute a set of statements as long as a condition is true.

```
# Ask for password until user enters correct secret:
secret = 'swordfish'
pw = ''
```

```
while pw != secret:
    pw = input("What's the secret word? ")

# Print count as long as it's less than 6:
count = 1
while count < 6:
  print(count)
  count += 1
```

Coding challenge:

- Given the example of while loop above;
- Limit the number of inputs to 3,
- Print a message to indicate remaining attempts,
- Warn the user at the last incorrect attempt. Good luck!

Some libraries of python may be called like random and time. No code is perfect at the begginning. Modify as you need. Google when you can't remember. Use cheat sheets.

```
import time
count = 5
while count > 0:
  print(f'Countdown {count}')
  time.sleep(1)
  count -= 1
print(10*'!' + 'Fire' + 10*'!')
```

For

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

```
# Print each fruit in a fruit list:
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
  print(fruit)

# Loop through the letters in the word "banana":
for char in "banana":
  print(char)
```

The built-in range() function

To loop through a set of code a specified number of times, we can use the range() function.

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
# Note that range(6) is not the values of 0 to 6, but the values 0 to 5.

# Using the range() function:
for my_num in range(6):
  print(my_num)


# The range() function defaults to 0 as a starting value, however it is possible
# to specify the starting value by adding a parameter: range(2, 6), which means
# values from 2 to 6 (but not including 6):

for my_num in range(2, 6):
  print(my_num)


# The range() function defaults to increment the sequence by 1, however it is
possible
# to specify the increment value by adding a third parameter: range(2, 30, 3):

for my_num in range(2, 30, 3):
  print(my_num)
```

Nested for loops

A nested loop is a loop inside a loop. The "inner loop" will be executed one time for each iteration of the "outer loop".

```
# Print each adjective for every fruit:
adjs = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for adj in adjs:
  for fruit in fruits:
    print(adj, fruit)
```

While vs For

Pushup analogy:

- For ----> Do pushup 10 times
- While --> Do pushup until I say stop!

Restaurant analogy:

- For ----> Bring 5 plates.
- While --> Bring food until I am full!

Transfer Statements

break

With the break statement we can stop the loop before it has looped through all the items.

```
# Exit the loop when x is "banana":
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
  print(fruit)
  if fruit == "banana":
    break
```

Another example using while:

```
count = 1
while count < 6:
  print(count)
  if count == 3:
    break
  count += 1
```

continue

With the continue statement we can stop the current iteration of the loop, and continue with the next:

```
# Do not print banana:
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
  if fruit == "banana":
    continue
  print(fruit)
```

return

The return statement in Python is a transfer statement that is used to exit a function and return a value to the calling code. When a return statement is executed inside a function, the function immediately stops executing and the program control is returned to the calling code, along with the value specified in the return statement.

The basic syntax of a return statement is as follows:

```
return [expression]
```

Here, expression is an optional expression that is evaluated and returned to the calling code. If no expression is provided, the return statement returns None by default.

It's worth noting that once a return statement is executed, the function stops executing and any code after the return statement is skipped. Therefore, if you have multiple return statements in a function, only the first one that is executed will return a value and terminate the function.

The pass Statement

In Python, the pass statement is a null statement that does nothing. It is used as a placeholder to indicate that a block of code is intentionally left blank or to be filled in later.

The pass statement is particularly useful in situations where you need to write code that does not do anything yet, but you still need to have some code in that location to satisfy the syntax requirements of the language. For example, you might use pass when defining a new function or class that doesn't yet have any implementation code.

Here is an example of how pass can be used in a function definition:

```
def my_function():
    pass
```

☺ **Happy Coding!** ✍

Clarusway