## Lecture 16

# Python Lists

### 8 Germinal, Year CCXXX

***Song of the day: Piano Concerto in G, II. Adagio assai** by Maurice Ravel (1931) performed by Alice Sara Ott and the WDR Sinfonieorchester Köln (2019).*

## Part 0: *Parameters and* `return` *review*

#### *Background*

Health points (HP) are one of the most important and ubiquitous elements of tabletop and video games, representing the amount of damage a user or a player can take before they lose a life or lose consciousness. Being as important as it is displaying a user's health in a way that is simple and effective is an important part of a video game's UI.

A **health bar** is one of the many ways to represent a player's health in video games. For example, we could "draw" a simple one in our shell / terminal by doing the following:

```
HP: 65 / 100
[=======   ]
```

**Figure 1**: A simple health bar, where the `'='` character represents one of the 10 "health bits" a player can have. Notice that each health bit here represents 10 HP.

This helps the player a ton with knowing how much health they have left, since just seeing the string `65 / 100` might lead the to believe that they have less (or more) health than they actually do; in general, visual representations are much more effective than numerical ones.

Another way of improving your UX experience is by adding colour to your UI. For example, if the user has a "high" amount of health, the health UI could be drawn in **green**, whereas when the player has a critically low amount of health, it could be drawn in **red**. We will be creating a program that executes both effects by breaking our code down into simple, short functions.

**NOTE**: If you're using IDLE, the colour feature might unfortunately not work. I will show examples below of how colouring appears in IDLE so that you know if what you are coding up is correct. Running this on any kind of terminal, including PyCharm's, should actually display colour.

#### *Your starting code*

In the **display_health.py** file, I have included the following lines of code to get you started:

```
from textColour import TextColour, colour
from random import randrange

GREEN = TextColour.GREEN
```

```
YELLOW = TextColour.YELLOW
RED = TextColour.RED
MIN_HEALTH = 0
MAX_HEALTH = 100

"""
WRITE YOUR FUNCTIONS BELOW
"""



"""
WRITE YOUR FUNCTIONS ABOVE
"""

def main():
    curr_health = randrange(MIN_HEALTH, MAX_HEALTH)
    # display_health(curr_health)  # uncomment to test


main()
```

**Code Block 1**: Your starting code. Since the `display_health()` function has yet to be defined, I have commented its invocation out inside the `main()` function. Feel free to uncomment it when you are ready to test it.

As usual, you do not need to worry about how any unfamiliar modules and constants work, just know that:

- The `colour()` function **accepts** an `str` and a `TextColour` (i.e. `GREEN`, `YELLOW`, and `RED`) as its two position arguments, and **returns** the contents of that string in the colour that was passed.

```
text = "Hello, world!"
print(colour(text, YELLOW))
```

Output:

```
Hello, world!
```

- `GREEN`, `YELLOW`, `RED` are `TextColour` objects that you can pass into the `colour()` function as arguments.
- `MIN_HEALTH` and `MAX_HEALTH` are, as their names imply, the minimum and maximum health our players will have.

Note that you **must** have the **textColour.py** file in the same directory in order for this program to work.

Please don't hesitate to raise your hand if you have any questions about how these work!

### *The `get_health_bar_length()` function*

Write a function `get_health_bar_length()` that accepts a single parameter: `health`, an integer that represents the user's current health. The same way we scaled `65 / 100` to seven health bits in **figure 1**, we will be scaling our own health bar. The only difference here is that, instead of scaling by a factor of 10, we will be scaling by a factor of 5 (that is, our health bar will measure one fifth of the user's current health).

Sample executions (feel free to pose these examples in your `main()` to test):

```
print(get_health_bar_length(100))
print(get_health_bar_length(67))
print(get_health_bar_length(44))
print(get_health_bar_length(29))
print(get_health_bar_length(2))
```

Output:

```
20
14
9
6
1
```

Note that the amount of health bits are being **rounded** up, so that a player with `2` HP will still display one remaining health bit.

### *The `get_health_bar()` function*

Your next function will accept one parameter: `health`, also an integer that represents the user's current health. This function will return a string that contains a health bar similar to the one in **figure 1**. You must, of course, use `get_health_bar_length()` inside of it in order to determine how many health bits it will contain. You should use the `'['` and `']'` characters to represent the beginning and end of the bar, and the `'='` character to represent a health bit.

Sample executions:

```
print(get_health_bar(100))
print(get_health_bar(67))
print(get_health_bar(44))
print(get_health_bar(29))
print(get_health_bar(2))
```

Output:

```
[====================]
[==========      ]
[=========           ]
[======              ]
[=                   ]
```

Notice that the second "section" of the health bar is shown as being "depleted" by using a space character `' '` instead of a `'='`; every health bar returned by `get_health_bar()` must be the same length.

### *The `get_colour_coded_string()` function*

This function will accept two position parameters:

- `health`: An integer representing the user's current health.
- `string`: A UI-related string (i.e. health bar, etc.)

And will return a colour-coded equivalent of the same `string` according to the player's current HP (all ranges are inclusive on both ends):

- `0 - 33` : **red**
- `34 - 66` : **yellow**
- `67 - 100` : **green**

Sample executions:

```
print(get_colour_coded_string(100, "Status"))
print(get_colour_coded_string(67, "Status"))
print(get_colour_coded_string(44, "Status"))
print(get_colour_coded_string(29, "Status"))
print(get_colour_coded_string(2, "Status"))
```

Output:

> Status

> Status

> Status

> Status

> Status

**NOTE**: If you're using IDLE, you might get the following output instead:

```
[92mStatus [0m
[92mStatus [0m
[93mStatus [0m
[91mStatus [0m
[91mStatus [0m
```

This is unfortunate, but just know that if you see the strings `[92m` (for green), `[93m` (for yellow), and `[91m` (for red), and the string `[0m` at the end of your output lines, you're doing this correctly. It's just another instance of IDLE being a buzzkill.

### *The `display_health()` function*

Finally, the `display_health()`, which will accept the `health` integer parameter as its only parameter, will use our other functions to print both a numerical and a visual health UI.

Sample executions:

```
display_health(92)
display_health(54)
display_health(6)
```

Terminal output:

```
HP: 92 / 100

[==================== ]
```

```
HP: 54 / 100

[===========          ]
```

```
HP: 6 / 100

[==                   ]
```

IDLE output:

```
[92mHP: 92 / 100 [0m
[92m[==================== ] [0m
[93mHP: 54 / 100 [0m
[93m[===========          ] [0m
[91mHP: 6 / 100 [0m
[91m[==                   ] [0m
```

**Solution**

## Part 1: *An Introduction to Python Lists*

So, we've seen sequences of numbers, and we've seen sequences of characters. There's not really anything else in the world of computing that gets more atomic than that—ones and zeroes are numbers, after all.

So instead of looking to smaller things, let's think the exact opposite way: numbers, characters, and really everything in Python are objects. So can we store a sequence of objects? SURE CAN.

```
CURRENT_YEAR = 2021

name = "Léa Seydoux"
age = 37
hometown = "Paris"
occupation = "Actress"
years_active = range(2005, CURRENT_YEAR + 1)

information = [name, age, hometown, occupation, years_active]
print(information)
```

Output:

```
['Léa Seydoux', 36, 'Paris', 'Actress', range(2005, 2022)]
```

Say hello to the Python list. In technical terms:

> **List**: A data type representing a sequence of objects. These objects (the elements of a list) can be of different types.

Here's another example of a list:

```
character = ["Link", 10, True, ["Kokiri Sword", "Razor Sword", "Gilded Sword"]]
```

**Code Block 2**: Even lists can be elements of a list.

## Part 2: *Lists as Sequences*

It also turns out that **indexing** works with the exact same syntax that we use for strings:

```
CURRENT_YEAR = 2021

name = "Léa Seydoux"
age = 36
hometown = "Paris"
occupation = "Actress"
years_active = range(2005, CURRENT_YEAR + 1)

information = [name, age, hometown, occupation, years_active]

print("{}, a(n) {} from {}".format(information[0], information[3].lower(), information[2]))
```

Output:

```
Léa Seydoux, a(n) actress from Paris
```

By way of a more useful example say that we have a list populated by integers and floats. How would we find the average of these numbers?

```
sample_list = [10, 50.6, -44, -0.0001, 56.00043, 45]

average = get_list_average(sample_list)

print(average)
```

Well, let's write the function stub first, before anything else:

```
def get_list_average(list):
    pass
```

As we learned last week, a list is a **sequence of elements**, the same way that a string is a sequence of characters and a range is a sequence of numbers. Therefore, we can iterate through it using a `for` -loop. How does this help us? Well, in order to

calculate the average of any set of numbers, we need both the sum of those numbers, and the amount of numbers in that set. We can get both of these things via a `for`-loop:

```
def get_list_average(list):
    total_sum = 0
    length = 0

    for number in list:
        total_sum += number
        length += 1

    average = total_sum / length
    return average
```

There's one thing we should be careful here, though. If the list that the user passes in as an argument turns out to be empty, our `average` calculation will yield a divide-by-zero error, so we should add a failsafe mechanism for that. Turns out that, just like with strings, we can use the `len()` function to get the length of a list. Let's use that to our advantage:

```
def get_list_average(list):
    length = len(list)

    if length == 0:
        print("WARNING: List is empty. Returning 0.0")
        return 0.0

    total_sum = 0

    for number in list:
        total_sum += number

    average = total_sum / length
    return average
```

**Code Block 3**: In this whole function, we never see list-specific behavior. We're just treating it like a sequence of numbers.

What does that mean for lists? If we can treat lists like sequences and, as we have seen, we can index them to get individual values, we can iterate through lists using indexing! Let's say we wanted to write a function that returns the **index of the largest number in a list of positive numbers**. By definition, we need to use indexing, so let's set up our function to do that:

```
def get_index_of_largest(list):
    length = len(list)

    for index in range(length):
        current_element = list[index]

        # Our algorithm
```

To find the largest number, we need to keep track of the current largest number, and if we encounter a number that is larger than the current largest number, we make the new number the current largest number. Since we made the assumption that the list will contain positive numbers only, our starting value can simply be `-1` or any other negative number, since we can safely assume that no positive number will be lower than it:

```
def get_index_of_largest(list):
    length = len(list)
    largest_number = -1
    largest_index = -1
```

```
    for index in range(length):
        current_element = list[index]
        if current_element > largest_number:
            largest_number = current_element
            largest_index = index

    return largest_index


def main():
    sample_list = [20, 45, 34.56432, 1, 2.3, 1.00002, 45.0000002, 5]
    largest_index = get_index_of_largest(sample_list)
    print("The index of the largest number in the list {} is {}.".format(sample_list, largest_index))


main()
```

Output:

```
The index of the largest number in the list [20, 45, 34.56432, 1, 2.3, 1.00002, 45.0000002, 5] is 6.
```

On the topic of indexing, it's also worth remembering that you can also index them the same way you would a string:

```
list = [1, 2, 3, 4, 5]

for index in range(len(list)):
    slice = list[:index + 1]
    print("Current slice: {}".format(slice))
```

Output:

```
Current slice: [1]
Current slice: [1, 2]
Current slice: [1, 2, 3]
Current slice: [1, 2, 3, 4]
Current slice: [1, 2, 3, 4, 5]
```

So, we can iterate and index through a list. In a similar fashion to `range()` and strings, we can also check for membership of elements using the `in` operator:

```
french_revolution_years = [1789, 1830, 1848]
year = 1967

print(year in french_revolution_years)
```

Output:

```
False
```

In other words, the value of `"1967"`, stored in the variable `year`, does *not* exist in the sequence (i.e. list) stored in the variable `french_revolution_years`.

Here's couple more examples:

```
>>> print(5 in [3, 6, [5, 7]])
False

>>> print(7 not in [1, 2, 3, 4])
True
```