Lecture 15

The `return` Statement

3 Germinal, Year CCXXX

Song of the day: In My Dreams *by Red Velvet (2022).*

I'm obsessed with their new EP sorry not sorry.

Part 0: *Good Vibrations*

Let's review what we learned about parameters on Monday by simulating how a smartphone registers a touch from a user. Depending on the type of touch, its direction, strength, and duration, the smartphone will give different types of **haptic feedback**.

Please use the file **haptic_feedback.py**.

Assume that there are four types of touches:

- **Single touch**: Represented by the `SINGLE` constant in our file.
- **Double tap**: Represented by the `DOUBLE` constant in our file.
- **Swipe**: Represented by the `SWIPE` constant in our file.
- **Hold**: Represented by the `HOLD` constant in our file.

If the user swipes, we can represent each cardinal direction using the constants `UP`, `DOWN`, `LEFT`, and `DOWN`. If the user **didn't** swipe, this direction automatically defaults to the constant `NO_DIRECTION`.

Duration and strength only matter in the case of a **hold**. In all other touch types, you can assume that the duration and strength will always both be `0.1`. Duration can be any positive integer, and strength can be any float from `0.0` (not inclusive) and `1.0` (inclusive).Duration and strength only matter in the case of a **hold**. In all other touch types, you can assume that the duration and strength will always both be `0.1`.

We will do this by creating three functions: `give_haptic_feedback()`, `register_touch()`, and our driver `main()` function.

`give_haptic_feedback()` will accept one positive numerical parameter called `touch_ratio`. Don't worry about what this value is just yet. Just know that:

- If the touch ratio is anywhere between `0.0` and `0.5` (non-inclusive on both ends), our function will print the message `"Vibrating once..."`.
- If the touch ratio is anywhere between `0.5` and `2.0` (inclusive on both ends), our function will print the message `"Vibrating twice..."`.
- If the touch ratio is anything higher than `2.0`, our function will print the message `"Vibrating thrice..."`.

`register_touch()` will accept four parameters: `touch_type`, `direction`, `duration`, `strength`.

- `touch_type` is a `TouchType` value that is represented by four of the nine constants described above (i.e. `SINGLE`, etc.).
- `direction` is a `SwipeDirection` value represented by the other five constants described above (i.e. `UP`, etc.). Remember that these constants are already included in your **file**, so you don't need to define them.
- You can assume that `duration` and `strength` will always be numerical values.

With these parameters, `register_touch()` will first check for the type of touch:

- If `touch_type` is equal to `SINGLE`, simply print the message `"Registering single touch..."`.
- If `touch_type` is equal to `DOUBLE`, simply print the message `"Registering double tap..."`.
- If `touch_type` is equal to `SWIPE`, print the message `"Registering single touch..."` and:
  - Print `"Exiting app..."` if `direction` is equal to `UP`.
  - Print `"Changing page..."` if `direction` is equal to `LEFT` or equal to `RIGHT`.
  - Print `"Scrolling up..."` if `direction` is equal to `DOWN`. You can assume that if `touch_type` is not equal to `SWIPE`, `direction` will be equal to `NO_DIRECTION`. You don't need to do anything else in this case.
- If `touch_type` is equal to `HOLD`, print the message `"Registering hold..."`, calculate the touch ratio (`strength / duration`), and invoke `give_haptic_feedback()` using the touch ratio as an argument.

Optional

The `main()` function will, as usual, drive the whole program. It should:

- **Always** ask what the touch type and...
- How strong it was.
- It should only ask for direction **if the touch type was *swipe***, and
- Only ask for touch duration **if the touch type was *hold***. Duration and strength only matter in the case of a **hold**. In all other touch types, you can assume that the duration and strength will always both be `0.1`.

You can assume that the user will always enter valid strings and numbers for these four values

Here are a few sample executions. Your format need not look the same:

```
What type of touch did the user perform? [single/double/swipe/hold] single
How strong was the user's touch? [0.0 to 1.0] 0.5
Registering single touch...
```

```
What type of touch did the user perform? [single/double/swipe/hold] double
How strong was the user's touch? [0.0 to 1.0] 0.3
Registering double tap...
```

```
What type of touch did the user perform? [single/double/swipe/hold] swipe
In what direction did the user swipe? up
How strong was the user's touch? [0.0 to 1.0] 0.7
Registering swipe...
Exiting app...
```

```
What type of touch did the user perform? [single/double/swipe/hold] hold
For how long did the user hold the touch? 1
How strong was the user's touch? [0.0 to 1.0] 0.25
Registering hold...
Vibrating once...
```

```
What type of touch did the user perform? [single/double/swipe/hold] hold
For how long did the user hold the touch? 2
How strong was the user's touch? [0.0 to 1.0] 1
Registering hold...
Vibrating twice...
```

Part 1: *The* `print()` *Function's Parameters and Default Parameters*

Speaking of parameters, what are the `print()` function's parameters? Since we'll be basically using `print()` for the rest of our programming lives, it stands to reason that we get as well acquainted with it as possible.

If we simply pass objects as arguments into the `print()` function, we know that it will print the values of those objects, separated by a single space (`" "`)

```
first_call = 3
second_call = 2
final_call = 1
start = 0
print(first_call, second_call, final_call, start)
```

Output:

```
3 2 1 0
```

So why does Python separate these arguments by a space? It turns out that the `print()` function has a "hidden" parameter called `sep` (short for separator) *whose default value is an single space*. Because this is the default the value, `print()` will take every argument we passed to it, and print them in order, separated by a space.

If we wanted to change the value of the `print()` function's separator, we would do something like this:

```
first_call = 3
second_call = 2
final_call = 1
start = 0
print(first_call, second_call, final_call, start, sep="...")
```

Output:

```
3...2...1...0
```

Here, we've told the `print()` function to replace its default separator for the string `"..."` . Notice that we have to explicitly type `sep=` in order to change the separator. Don't forget to do so— `print()` will just assume that it is another parameter to `print()` .

To generalize this behavior, here's what the `print()` function's definition's signature probably looks like:

```
def print(object1, object2, …, sep=' '):
    # more code
    # more code
```

Giving a parameter a value in the function signature gives it a **default** value. These are useful when you have functions that give the user the option of entering their own value if they choose to do so. If they don't, the function will use the default parameter:

```
def make_french_fries(number_of_orders, uses_air_frier=False):
    if uses_air_frier:
        print("Making", number_of_orders, "french fries orders using an oil frier.")
    else:
        print("Making", number_of_orders, "french fries orders using an air frier.")

def main():
    make_french_fries(10)
    make_french_fries(7, True)
    make_french_fries(3, uses_air_frier=True)
    make_french_fries(number_of_orders=20, uses_air_frier=False)

main()
```

Code block 1: These are all valid function calls for `make_french_fries()`.

Output:

```
Making 10 french fries orders using an air frier.
Making 7 french fries orders using an oil frier.
Making 3 french fries orders using an oil frier.
Making 20 french fries orders using an air frier.
```

We'll revisit `print()` to talk about more of its optional parameters, but this is all we need for the time being.

Part 2: *The* `return` *Statement*

Remember how the `range()` function accepted arguments for its start, stop, and step values, and then gave us back a sequence based on those numbers?

```
def main():
    zero_to_ten_evens = range(0, 11, 2)

    print("The numbers inside this sequence are...")
    for number in zero_to_ten_evens:
        print(number)

main()
```

Output:

```
The numbers inside this sequence are...
0
2
```

```
4
6
8
10
```

That is, we set the variable `zero_to_ten_evens` to `range(0, 11, 2)`, but the actual value that this is equivalent to is a sequence from 0 to 11 with a step of two.

This is one way this can be said in English:

> The function `range()`, when given the arguments `0`, `11`, and `2`, ***returns*** a sequence from 0 to 11 with a step of two to the variable `zero_to_ten_evens`.

This is certainly not the only built-in Python function that we have seen that shows this behavior. Consider the `input()` function; in English, you would say:

> The function `input()`, when given user input as an argument, ***returns*** the string equivalent of that user input.

> The function `int()`, when given a string containing an integer as an argument, ***returns*** the `int` equivalent of that string input.

You can probably see where this is going: functions can return values. So far, all of our functions have either works in isolation of the outside world (i.e. do not accept parameters), or accept data from the outside world through parameters and then use that data inside the function (see code block 2 for an example of this).

We've thus far only printed the results of our functions, but more often than not, you'll want your functions to actually ***return*** the result to the outside world so that you can use that result in other unrelated calculations. (the `input()` function would be pretty useless if all it did was print our input again, and not actually let us save it in a variable).

Let's modify the code from code block 2 to return the result of out volume calculations:

```
PI = 3.1415


def get_cylinder_volume(radius, height):
    """
    Calculates the volume of a cylinder of a given radius and height.

    :param radius: A float or integer representing the radius of the bases
    :param height: A float or integer representing the height of the cylinder
    :return: volume: A float representing the volume of our cylinder.
    """
    volume = PI * (radius ** 2) * height

    return volume


def main():
    sample_radius = 5
    sample_height = 1.4
    get_cylinder_volume(sample_radius, sample_height)


main()
```

That's it. Notice though, that our doctstring also changed, because now we're actually returning something, so by convention we usually write the name of the returned variable, and a short explanation.

Code block 2: `print_cylinder_volume()` is now `get_cylinder_volume()`.

Now, if we ran this, nothing would show up in our Terminal screen. Why? Because we're no longer printing anything. `get_cylinder_volume()` calculates the volume of our cylinder and returns it. The problem is that we're not saving that returned value anywhere. Exactly the same way that we need to save the returned value from the `input()` function, we need to save the returned value from `get_cylinder_volume()` in a variable in order to use it:

```
PI = 3.1415


def get_cylinder_volume(radius, height):
    """
    Calculates the volume of a cylinder of a given radius and height.

    :param radius: A float or integer representing the radius of the bases
    :param height: A float or integer representing the height of the cylinder
    :return: volume: A float representing the volume of our cylinder.
    """
    volume = PI * (radius ** 2) * height

    return volume


def main():
    sample_radius = 5
    sample_height = 1.4
    cylinder_volume = get_cylinder_volume(sample_radius, sample_height)  # our result is being stored here

    print("The volume of the cylinder is", cylinder_volume)  # our result is being used here

main()
```