User input, functions, loops, conditional statements

Learning outcomes

- Create and query an array (also known as a list)
- Write an iterative statement (a "for" loop)
- Use logic and Boolean operators
- Write a conditional ("if/else") statement

Recap from last session

- What is a string?
- What is a variable?
- What is a formatted string?
- How do you print a formatted string to your terminal?
- What are the two number types we learned about last week and what’s the difference between them?

Let's get started!

Lists (also called arrays)

Just like a shopping list, you can use a list in Python to store items. You can create a list just like you create a variable:

```
empty_list = []
list_of_numbers = [ 0 , 1 , 2 , 3 , 4 , 5 ]
my_favourite_fruits = ["pineapples", "oranges", "bananas"]
mixed_list = [ 15 , "sunshine", "jumper", 4 , "sky"]
```

Here are some things to remember about lists:

- List items go between square brackets
- Items in a list are separated by commas
- Strings in a list must be in quotation marks

You can access an item in a list by using its name and its index number, like this:

```
my_favourite_fruits[0]
```

Try to print the above and see what happens. Is that what you expected? In the world of programming, the first item in a list is always at position 0 (its **index** position). This means that `[0]` is the first item in the list, `[1]` is the second item, and so on.

Task:

1. Create a shopping list called `shopping_list` and store at least 5 items you want to buy.
2. Print the 1st item (index 0), 3rd item (index 2) and 5th item in your terminal.

You can also access the last item of the list without knowing the exact index. For example, if you want to access `bananas` in the `my_favourite_fruits` list created above, you could do so with:

```
my_favourite_fruits[2]
```

But what if the list is very large (say index 98) and you don't know what the last index is? You can also access `bananas` by calling

```
my_favourite_fruits[-1]
```

which returns the last element in the list, regardless of the size of the list.

It's also possible to access a range of items by specifying the starting and ending index. For example, `my_favourite_fruits[0:2]` will return `pineapples` and `oranges` . As you might notice, the ending index is not inclusive (the item at index 2 is not included).

Task:

1. Call `my_favourite_fruits[-2]` and see the output.
2. What happens if you call `my_favourite_fruits[5]` or `my_favourite_fruits[-10]` ?
3. Try to guess the output of `my_favourite_fruits[0:-1]` before you run the command.

It is also useful to know how many items are in the list. Python has a built-in function called `len()` to count the number of items in the list. Using the same example `my_favourite_fruits` , you can wrap it with `len()` as follows:

```
print(len(my_favourite_fruits))
```

which prints `3` because there are 3 items in the list. `len()` would be useful if you have a very large list (say, 1000 items) and you are not sure how many items are in the list (counting manually would take too long).

Appending and removing items from a list

You can easily concatenate multiple lists with `+` operator. For example, if there are two lists you want to combine, you can do so as follows:

```
my_favourite_fruits = ["pineapples", "oranges", "bananas"]
your_favourite_fruits = ["pineapples", "apples"]

new_fruit_list = my_favourite_fruits + your_favourite_fruits
print(new_fruit_list)
```

which will print `["pineapples", "oranges", "bananas", "pineapples", "apples"]` .

Other ways to modify a list are using Python's built-in methods, which allow you to dynamically append and remove an item from a list. Let's start with `append` . You can append a new item to an existing list with `append()` as follows:

```
animals = ['cat', 'dog', 'rabbit']
animals.append('dolphin')
print("new animal list is:", animals)
```

which will print `new animal list is: ['cat', 'dog', 'rabbit', 'dolphin']` . As you can see, `append()` appends an item to the last index of the list.

If you want to add an item anywhere other than the last index, you can do so as follows:

```
animals = ['cat', 'dog', 'rabbit']
animals.insert(0, 'koala')
print("new animal list is:", animals)
```

The output will be `new animal list is: ['koala', 'cat', 'dog', 'rabbit']` .

Task:

1. Using the same `animals` list, add 3 more animals.

2. Using the same `animals` list, what happens if you try to do `animals.insert(100, 'dragon')` ?
3. What happens if you do `animals[0] = 'myself'` ?

Similar to adding an item, you can remove an item from the list. The first built-in method is `pop()`

```
animals = ['cat', 'dog', 'rabbit']
removed_animal = animals.pop()
print(removed_animal, " has been removed")
print("new animal list is:", animals)
```

The first `print` will print `rabbit has been removed`, and the second `print` will print `new animal list is: ['cat', 'dog']`.

If you want to remove an item other than the last item in the list, you can call `remove(item)` as follows:

```
animals = ['cat', 'dog', 'rabbit']
animals.remove('dog')
print("new animal list is:", animals)
```

which will print `new animal list is: ['cat', 'rabbit']`.

If you want to remove all the items in the list, `clear()` will clear all the elements in the list.

```
animals = ['cat', 'dog', 'rabbit']
animals.clear()
print("new animal list is:", animals)
```

which will print `new animal list is: []`.

Task:

1. Using the same `animals` list, call `animals.pop(2)` and see what happens.
2. Try to call `pop()` to an empty list.

You can do much more than what we have just learnt with a list, such as `extend`, `reverse`, `sort` etc. See https://developers.google.com/edu/python/lists for details.

Nested list

A list can have another list, and this is called nested list. If you work on data you will often see the nested arrays, so it is good to get used to this data structure.

```
nested_animals = ['human', ['cat', 'dog', 'rabbit'], ['dolphin', 'turtle', 'frog']]
```

Accessing a particular item is a bit tricky, but it is possible using two indices, the first index for the outer list and the second index for an inner list.

```
print(nested_animals[1][1])
```

which will return `dog`.

Nested lists are far more advanced, so do not worry if it is overwhelming. Practice with the nested lists so that you will get used to handling them.

Task:

1. Create a 3-by-3 nested list called `square` with items starting from 1 to 9.
2. How do you access the middle row in the nested list ? The output should be `[4,5,6]`

3. How do you access the middle item in the nested list ? The output should be `5` .

Using "logic" in programming

Logic is an important concept in computer programming. We can write programs that can take a specific action based on whether a condition is satisfied ("True") or not satisfied ("False").

An example we use every day is signing into a service. If both your username and password match what's in the service's database, you're logged in. If they don't, you're not allowed access.

Here's a list of logic operators and what they "evaluate":

Operator	Description
`and`	Are both/all conditions true
`or`	Is at least one condition true
`not`	negation
`!=`	not equal to
`==` (double equals)	equal to
`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to
`True`	It is true
`False`	It is false

Just like in maths class, if you have things inside of brackets to evaluate, deal with the things inside the brackets first:

Statement to evaluate	Answer
`1 == 1`	`True`
`1 == 0`	`False`
`(1 == 1) and (1 == 0)`	(True) and (False) → `False`

Task:

1. Write the following Python code and see the output (try to guess before you run it).

```
x = [1,2,3]
y = [2,4,5]
print(x == y)
```

2. Write the following Python code and see the output (try to guess before you run it).

```
x = [1,2,3]
y = [2,4,5]
print(x != y)
```

3. Write the following Python code and see the output (try to guess before you run it).

```
x = [1,2,3]
```

```
y = [2,4,5]
print(len(x) == len(y))
```

4. Write the following Python code and see the output (try to guess before you run it).

```
print(1 * 10 == 10)
```

5. Write the following Python code and see the output (try to guess before you run it).

```
print((1*10 < 9) and (2 + 3 == 5))
```

6. Write the following Python code and see the output (try to guess before you run it).

```
print((1*10 < 9) or (2 + 3 == 5))
```

"For" loops

"For" loops are also called iterative loops, because they iterate (run through their code repeatedly) until there's nothing left to iterate through.

This concept makes more sense if we go back to what we learned about lists earlier.

Imagine you're going to a coding party with some friends from class, and you need to go to the supermarket to get supplies. Keeping with the Python theme of your party, you bring your laptop with you, make a digital "shopping cart" in Python, and add stuff to it using `.append()` , which adds an item to the list.

Before you get to the checkout counter, you want to print the contents of your cart, to make sure you have everything. Here's how you'd do that in Python using a for loop:

```
my_shopping_cart = ["cake", "plates", "plastic forks", "juice", "cups"]
for item in my_shopping_cart:
    print(item)
```

The only condition the Python interpreter cares about in your for loop is whether there's anything left to iterate through. Since you've only got 5 items in your shopping cart, your loop will only run 5 times. Python's clever that way.

A few helpful things about for loops:

- The code that's part of your for loop must be indented. Make sure you don't forget the colon at the end of the first line!
- For loops don't return anything, but simply iterate though each item
- This is the format that you need to use when you write a for loop:

```
for x in name_of_thing_to_iterate_through:
    print(x)
    # or do something else
```

where `x` can be called anything – you could call it `x` , or `bananas` , or `count` , or `item` – it's basically just a placeholder name. But the name of the thing you're iterating through is important – it's the name of your list, so that Python knows what to iterate through.

You can do lots of things with for loops, including adding items to lists, deleting items from lists or from files, etc.

Conditional statements ("if" statements)

We talked about logical operators a bit earlier and how they're useful, because we can use them to write programs that can take a specific action based on whether a condition is satisfied or not.

You can think of each conditional statement as a different "branch" or "path" of code that you can follow – a bit like a fork in the road, you have to choose one path to follow because you can't be on both at the same time.

If you have a conditional statement that evaluates to `True`, then Python will execute the block of code that immediately follows the if statement. Where the statement evaluates to `False`, Python will skip that block of code. Just like in a for loop, the code you want to run when a condition is `True` must be indented, and preceded by a ":".

Task

1. Copy and paste the code below into a new file called `numbers.py` and run the file.

```
number = input( "Enter a number between 1 and 10: " )
number = int(number) # Converts the input string to an integer
if number > 10 :
    print("Too high!")
if number <= 0 :
    print("Too low!")
```

1. What happens if you enter a number that's greater than 10?
2. What happens if you enter a number that's less than 0?
3. What happens if you enter a number between 1 and 10?
4. What happens if you remove the line with `number = int(number)` ?

If none of your conditions are met, Python will simply carry on through the rest of your code. In the example above, we didn't write any code for the situation where the user enters a number between 1 and 10, so you didn't see any sort of helpful message or feedback in your terminal.

Notice how we used `int()` to convert the user input into a number. `input()` treats all input as a string, so we used information in Python's documentation to figure out how to convert the user input to a number. You don't need to worry about this right now, but it's an example of something useful you can do with Python.

If else & else statements

An else statement is like a catch-all for when none of your conditions are satisfied. That means you can only have one else statement, and it can't exist on its own without an if statement. If you think you need more else statements, you actually need more if statements!

Using else statements isn't mandatory, but they're handy for debugging because they can help to catch unexpected behaviour in a program.

Example of using if/else statement is as follows:

```
is_morning = False

if is_morning:
    print("Good morning")
else:
    print("Hello")
```

When you run this program, it will print `Hello` because `is_morning` evaluates to `False` .

Task

Using the code we just looked at above:

1. Add an if statement that prints a message if you enter a number between 1 and 10 (just like the previous exercise).
2. Add an else statement to print `Okay` if the number is within 1 and 10.

You can have more than 2 branches using the `elif` statement. The basic format is as follows:

```python
time = 1
if time >= 0 and time < 12:
    print("Good morning")
elif time >= 12 and time <= 17:
    print("Hello")
elif time >= 18 and time <= 24:
    print("Good evening")
else:
    print("Time has to be between 0 and 24")
```

Similar to nested lists in the previous section, you can create more complex logic trees with nested conditional statements. You do not need to worry about this for now, but if you are curious, here is an example of a nested conditional statement.

```python
is_hungry = True
had_meal = False

if is_hungry:
    if had_meal:
        print("I am still hungry, but I shouldn't eat more...")
    else:
        print("I am going to a restaurant")
else:
    print("I am going to sleep")
```

Homework

1. Review the examples from today's session to make sure you understand them. If you get stuck, don't be shy about asking an instructor for help!
2. Write a Python program which iterates through the integers from 1 to 50.
    - For multiples of three, print "Fizz" instead of the number,
    - For multiples of five, print "Buzz".
    - For numbers which are multiples of both three and five, print "FizzBuzz".

The sample output:

```
1
2
Fizz
4
Buzz
...
13
14
FizzBuzz
...
```

3. Write a Python program to print the following pattern, using a nested for loop.

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

If you want to practice more, there is a website called Codewars, where you have a number of programming exercises, from easy ones to very advanced questions.

Reference

The content of this course is developed based on the following references:

- CFG Advanced Python course
- w3resource