Lecture 0 - Part 3: Comprehensions and Functions

1 Comprehensions

- Comprehensions are a concise and powerful way to create new mutable object (lists, sets, and dictionaries) from a string or any other iterable container. They allow you to create a new collection by iterating over an existing one and applying a transformation or filtering operation.
- There exist three comprehensions: list comprehension, set comprehension, and dictionary comprehension. The syntax of the comprehension is given by:

```
# List comprehension
[<expr> for <loop var> in <iterable>]
# Set comprehension
{<expr> for <loop var> in <iterable>}
# Dictionary comprehension
{<key>: <value> for <loop var> in <iterable>}
```

1.1 For-loop vs comprehension:

A for-loop can be used to iterate over a sequence and apply a transformation or filter operation. However, using a comprehension is often more concise and easier to read. Here's an example of creating a new list of squared numbers using a for-loop:

```
numbers = [1, 2, 3, 4, 5]
squares = []
for n in numbers:
    squares.append(n ** 2)
```

The same thing can be achieved using a list comprehension:

```
numbers = [1, 2, 3, 4, 5]
squares = [n ** 2 for n in numbers]
```

1.2 Conditionals in List Comprehension:

You can also add conditional statements to list comprehensions to filter the items being transformed. Here's an example of creating a new list of even numbers using a for-loop:

```
numbers = [1, 2, 3, 4, 5]
evens = []
for n in numbers:
    if n % 2 == 0:
        evens.append(n)
```

The same thing can be achieved using a list comprehension:

```
numbers = [1, 2, 3, 4, 5]
evens = [n for n in numbers if n % 2 == 0]
```

1.3 Nested IF with List Comprehension:

You can also use nested if statements in list comprehensions to create more complex filters. Here's an example of creating a new list of even numbers that are greater than 2 using a for-loop:

```
numbers = [1, 2, 3, 4, 5]
evens_greater_than_2 = []
for n in numbers:
    if n % 2 == 0:
        if n > 2:
            evens_greater_than_2.append(n)
```

The same thing can be achieved using a list comprehension with nested if statements:

```
numbers = [1, 2, 3, 4, 5]
evens_greater_than_2 = [n for n in numbers if n % 2 == 0 if n > 2]
```

1.4 `if-else` statement with List Comprehension

If-else statements can also be used in list comprehensions to conditionally include elements based on some condition. Here's an example that creates a list using `if-else` statement:

```
squared_even_numbers = []
for num in numbers:
    if num % 2 == 0:
        squared_even_numbers.append(num ** 2)
    else:
        squared_even_numbers.append(num)
```

How if we have Nested `if-else` statement?

1.4.1 Nested Loops in List Comprehension

The nested loops list comprehension syntax is:

[ ]:
```
numbers=[1,2,3,4,5,6]
squared_even_numbers = [ num**2 if num%2==0 else num**3 if num%3==0 else num for␣
↪num in numbers]
squared_even_numbers
```

2 Functions

In our real life, to solve a big problem we have to break it into smaller ones.

Programming languages kept the same principle; As our program grows larger and larger, Functions help break it into smaller and modular chunks.

2.1 What's a function?

A function is a block of organized, reusable code that is used to perform a single, related action.

2.2 Why functions?

Functions make programs more organized and manageable. Furthermore, it avoids repetition and makes the code reusable. ## Function syntax The function syntax is:

```
def functionname( parameters ):
    "function_docstring"
    function_instructions
    return [expression]
```

2.3 How to define a function?

Here are simple rules to define a function in Python:

- Function blocks begin with the keyword `def` followed by the function name and parentheses ( ).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- The code block within every function starts with a colon : and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return `None`.
- By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

For example, the following function `printname` take a name as a parameter and print "Hi, <name>, How was your day?". It return `None`

```
    def printname(name):
        """This function says Hi"""
        print("Hi, ",name, "How was your day?")
        return
```

2.4 How to call a function

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printname function

```
def printname(name):
        """This function says Hi"""
        print("Hi, ",name, "How was your day?")
        return
    printname("Ahmed")
    printname("Ali  ")
```

2.5 The return statement

The return statement is used to exit a function and go back to the place from where it was called.

This statement can contain an expression that gets evaluated and the value is returned. If there is no expression in the statement or the `return` statement itself is not present inside a function, then the function will return the `None` object.

```
    print(printname("Ahmed"))
```

That will return `None`

Exercise

- Write a function that return the square of a number.
- Write a function that return the absolute value of a number without using `abs()`

2.6 Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.

The lifetime of a variable is the period throughout which the variable exists in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Here is an example to illustrate the scope of a variable inside a function.

```
def mark():
    m = 10
    print("Value inside function:",m)

m = 20
mark()
print("Value outside function:",m)
```

- In the previous example, we can see that the value of m is 20 initially. Even though the function mark() changed the value of m to 10, it did not affect the value outside the function.
- This is because the variable m inside the function is different (local to the function) from the one outside. Although they have the same names, they are two different variables with different scopes.
- On the other hand, variables outside of the function are visible from inside. They have a global scope.

- We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword global. ## Function Arguments

You can call a function by using the following types of formal arguments:

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

2.7 Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

You definitely need to pass the same number of arguments as in definition, otherwise it gives a syntax error.

```
def infos(name,age):
        print("my name is:", name)
        print("I am",age,"years old")
    infos("Ahmed")
    infos("Ahmed",20)
```

2.8 Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

```
def infos(name,age):
        print("my name is:", name)
        print("I am",age,"years old")
    infos(age=20,name="Ahmed")
```

2.9 Default arguments (optional)

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

```
def infos(name,age=19):
        print("my name is:", name)
        print("I am",age,"years old")
    infos(age=20,name="Ahmed")
    infos(name="Ahmed")
```

2.10 Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

An asterisk * is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

2.11 Variable-length arguments

```
    def infos(name,age,*marks):
        print("my name is:", name)
        print("I am",age,"years old")
        for i in range(len(marks)):
            print("my",i+1,"th mark is", marks[i])
    infos("Ahmed",20)
    infos("Ahmed",20,19,10,5)
    infos("Ahmed",20,19,10,11,17,5)
```

2.12 The Anonymous Functions (Lambda functions)

These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression. ## Syntax

The syntax of lambda functions contains only a single statement, which is as follows:

```
func_name=lambda [arg1 [,arg2,.....argn]]:expression
```

Following is the example to show how lambda form of function works:

```
special_sum=lambda a,b: 2*a+3*b
    print(special_sum(1,1))
    print(special_sum(2,9))
```

2.13 Recursive Functions

A recursive function is one that calls itself. ## Recursive Definitions Every recursive function definition includes two parts:

- **Base case(s) (non-recursive):** One or more simple cases that can be done right away
- **Recursive case(s):** One or more cases that require solving "simpler" version(s) of the original problem.

For example the factorial function can be defined recursively as

```
def fact(n):
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

2.14 Recursive function good or bad ?

- Simpler, more intuitive: For inductively defined computation, recursive algorithm may be natural and close to mathematical specification.
- Easy from programming point of view.
- May not efficient computation point of view.

2.15 Examples

- Write a recursive function that compute the product of $a * b$.
- Write a recursive function that compute the gcd of a and b.
- Write a recursive function that compute Fibonacci numbers.
- Write a recursive function that compute x to the power n; (x^n).