

Unit - 4

Functional Programming

is the use of functions to make our program smaller to type and more efficient.

The `Map()` Function

Q1. Write a python program using a function which takes list of strings as an input and returns length of each string present in the list

Q2. Write a python program, with the same input as above but it returns the uppercase of each string.

Q3. Given list of integers, return squares of each element

- All of these applications can be solved with a one liner using the `map()` function.
- Syntax: `map(<callable_function>, <iterable_objects>)`
- Input: '*n*' elements
- Output: '*n*' elements
- For all of the above questions, I will comment out the traditional approach and make a `map()` function instead.
- Example:

```
givenStringList = ["hello", "he", "abcde", "ooooooooo"]
```

PYTHON

```
# MAP FUNCTION
```

```
print(list(map(len, givenStringList)))
```

Q4. Write a python program using functions that displays table. Given n

The `lambda` function

- Anonymous function
- One liner
- syntax: `lambda <args>: <expression>`
- The expression is what gets returned.
- For example:

```
func = lambda x: x*2
```

```
# is equivalent to:
```

```
def func(x):  
    return x*2
```

PYTHON

- We usually use a lambda function when another function expects a function as a argument. For example the `map()` function. We do not assign it to a `func` object like we did above.
- Something like this:

```
print(list(map(lambda x: x+1, l1)))
# adds 1 to every element of l1
```

Q5. Write a lambda function that takes 2 args and multiplies them, display the result.

Q6. Write a lambda function to add elements of 2 lists and display the result.

The `filter()` function

- Similar to a `map` function in terms of syntax: `filter(<callable_function>, <iterable_objects>)`
- The `callable_function` should return a true or false.
- The elements of the `iterable_object` will make it on to the final result only if the `callable_function` returns true for that element.
- Input: 'n' elements
- Output: 0 to 'n' elements
- The callable function could be a lambda function too.
- For example, this is a program to make a list of only even numbers from a list of numbers:

```
l = [1, 2, 3, 4, 5, 6, 7, 8]
print(list(filter(lambda x: x%2==0, l)))
# prints [2, 4, 6, 8]
```

Q7. Write a python program that gets even numbers from a given list

Q8. Write a python program where input is a list of strings and the return a list of strings which contain the letter 'r'

Q9. Write a python program to find the difference between two lists using filter function.

The `reduce` function

- You have to `import functools`
- Syntax: `functools.reduce(callable, iterable)`
- Input: 'n' elements
- Output: 1 element
- used to **apply a particular function passed in its argument to all of the list elements** mentioned in the iterable passed along.
- **Working :**
 - At first step, first two elements of sequence are picked and the result is obtained.
 - Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.
 - This process continues till no more elements are left in the container.
 - The final returned result is returned.
- Example:

```
# importing functools for reduce()
import functools
```

```
lis = [1, 3, 5, 6, 2]

# using reduce to compute sum of list
print("The sum of the list elements is : ", end="")
print(functools.reduce(lambda a, b: a+b, lis))
```

Output:

```
The sum of the list elements is : 17
```

SHELL

Q10. Write a program that takes input as a list of strings as input and output is the list of strings whose length is less than 4 in uppercase.

Q11. Write a python program which takes list of number and returns the sum of all elements. Use the reduce function but not a lambda function.

List Comprehension

- A Python list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element in the Python list.
- It is a concise way of making a list <= One liner

Advantages of List Comprehension

- More time-efficient and space-efficient than loops.
- Require fewer lines of code.
- Transforms iterative statement into a formula.

Syntax of List Comprehension

```
newList = [expression(element) for element in oldList if condition]
```

PYTHON

Key Points

- Comprehension of the list is an effective means of describing and constructing lists based on current lists.
- Generally, list comprehension is more lightweight and simpler than standard list formation functions and loops.
- We should not write long codes for list comprehensions in order to ensure user-friendly code.
- Every comprehension of the list can be rewritten in for loop, but in the context of list interpretation, every for loop can not be rewritten.

Examples:

```
list = [i for i in range(11) if i % 2 == 0]
print(list)
```

PYTHON

- Output:

```
[0, 2, 4, 6, 8, 10]
```

- One more example: Matrix using List comprehension:

```
matrix = [[j for j in range(3)] for i in range(3)]  
print(matrix)
```

PYTHON

- Output:

```
[[0, 1, 2], [0, 1, 2], [0, 1, 2]]
```

Q12. Write a python program using list comprehension which takes a list of words as input and prints the last letter of each word.

Q13. Write a python program using list comprehension that combines the elements of 2 lists if they are not equal.

Q14. Write a python program that finds common numbers from 2 lists

Q15. Write a python program that prints the string and string length of each string in a list

Q16. Write a python program to find all words whose length is more than 5 and less than or equal to 10.

Q17. Input: list of numbers | Output: numbers divisible by both 2 and 3.

Q18. Input: list of numbers | Output: Even and odd for every number.

Modules

- A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables.
- Whenever we do either `import math` or `import random`, here, `math` and `random` are called *modules*

Why modules?

- You can reuse code written in other files. Similar idea of functions but you can use stuff from other files too.
- This follows that all advantages of functions are advantages of modules too:
 - Code reusability
 - Code readability
 - Code maintainability
 - etc.,

The different ways of importing modules

1. `import <module name>` : The classic vanilla way of importing the module. If you want to use a function from this module, you have to use `module_name.function()`
2. `import <module name> as <short name>` : The alias way. You give a shorter name to your module and use that to reference the module. So while calling functions, it would be `short_name.function()`

3. `from <module name> import <functions>` allows you to import only the function you need from a module. Imagine you only want to use `randint()` from the `random` module. Importing `random` as a whole would just make your code less efficient as it has to store all functions of `random`. Instead, you can use `from random import randint`. Note that in this way, you don't need to reference `random` while calling function. You do not need to do `random.randint()`, instead, you'll just be calling `randint` directly: `randint()`
4. `from <module name> import *` imports all functions from a particular module. The `*` signifies 'all', similar to SQL if you have studied that. The advantage of this way of importing is that it allows you to call your functions from that module directly. If you want to use `randint`, you can directly call `randint()`. This method is not recommended unless absolutely necessary. Imagine you have another module with the same function name, and you import that module using the same way as well. The names of the functions would clash. This is called a namespace clash. This might not cause problems on a smaller project but while coding a larger project, with multiple modules, the name of the functions from 2 modules would clash and debugging that would be a pain.

How do you make your own module?

- Just make a .py file. Seriously. Just make your own .py file in the same directory as your `main.py` file. Let me explain with an example:

PYTHON

```
# This is the contents of modulefile.py
def add(a, b):
    return a+b
```

This would be my module. It has a function called `add` that takes 2 arguments and returns the sum. Now lets make our main file:

PYTHON


```
import modulefile
print(modulefile.add(2,3))
```

This calls the `add` function from `modulefile` and prints the output, that is `6`

- It is important for both these files to be in the same directory or folder. Or the import will not work properly, you will need to give a proper path if your module file is in another folder.

The `__doc__` method

This is a method/function of most inbuilt functions. It's job is to print the `documentation` of the module. Example, `print(math.__doc__)` gives the documentation of the `math` module.

 Documentation is one of the most important parts of programming. It provides information on how a particular thing works. It's like a manual. Sometimes it's called a manual too. Tools like Qt, tkinter, modules like math, random, etc, all have documentation. There is a popular phrase for beginners asking beginner questions to experienced programmers, `RTFM`, it stands for `Read The F*cking Manual`

How do you differentiate between a module python file and an executable python file?

We saw earlier that you just make a python file for a module. Now how do you differentiate between a normal python file that is supposed to be run and a python file that is supposed to be used as a module?

- We use this thing called a namespace. Executing `__name__` gives us the namespace of the place it's being executed at. For example, executing `__name__` outside every function, aka in the global scope, returns `'__main__'`. We use this thing to differentiate between a exec file and a module file.
- We protect the code in our exec program with an if statement:

```
PYTHON
# file that is to be run

def main():
    # our main program code goes here
    print(''This gets executed if this code is being run directly BUT will not
run if this file is just imported. However, if you import this file and then
call the main() function in specific, then this would be executed'')

if __name__ == '__main__':
    main()
```


We are checking if the namespace is `__main__` and then executing the function called `main()` This is just a convention and you can call your main function anything.

A file that is supposed to be a module will **NOT** have an `if __name__ == '__main__':`

There might be a tricky question asked wrt this on ISA. Just go through this program, it's pretty descriptive:

```
PYTHON
print ("Always executed")

if __name__ == "__main__":
    print ("Executed when invoked directly")
else:
    print ("Executed when imported")
```

 The following programs must be written in the form of a module. I am making module files with a descriptive name on my GitHub Repository. The question name will not be there as module names cannot start with a number :(The main file that imports all of these functions and executes them is called `mainModules.py`

Q19. Write a python script to remove all numbers from a list that are divisible from 3 *Module Name In Repo: **remNumbersBy3.py***

Q20. Takes a range, find numbers that are both odd and palindromic. *Module Name In Repo: **oddAndPalindromic.py***

Q21. Write a program to create a list of numbers and a list of strings of same size and combine them to make a list of tuples *Module Name In Repo: **numberStringCombination.py***

Q22. Write a program to determine the max element in a list *Module Name In Repo: **maxElement.py***

Q23. Write a python program to form a list of strings made of first and last character of the string if the length of the string is greater than or equal to 3. *Module Name In Repo: **firstLastChar.py***

Non module questions:

Q24. Count number of spaces in a string

Q25. Form a list of strings made of the first 2 and last 2 characters from a given list of words if the length of the string is greater than 4

Q26. Write a python program to find and display the intersection of 2 lists

Testing and Debugging

- *Debugging* is identifying and removing errors in a program.
- *Testing* is a process of testing our software to see if it meets expected requirements

Types Of Testing:

1. **Black Box Testing** is a type of software testing in which the functionality of the software is not known. The testing is done without the internal knowledge of the products.
2. **White Box Testing** techniques analyse the internal structures the used data structures, internal design, code structure, and the working of the software rather than just the functionality as in black box testing. It is also called glass box testing or clear box testing, structural testing, transparent testing or open box testing.
3. **Grey Box Testing** is a software testing technique which is a combination of Black Box testing technique and White Box Testing technique. In Black Box Testing technique, tester is unknown to the internal structure of the item being tested and in White Box Testing the internal structure is known to tester. The internal structure is partially known in Grey Box Testing. This includes access to internal data structures and algorithms for purpose of designing the test cases.

The Pdb Debugger

- Full form: Python Debugger
- Its a debugger built in to Python
- You can run it from the terminal/Command Prompt using the command:

```
python3 -m pdb myscript.py
```

SHELL

pdb supports:

- Setting breakpoints
- Stepping through code
- Source code listing
- Viewing stack traces

Commands:

Command	Function
help/h	To Display all commands
where	Displays the stack trace and line number of the current line
next/n	Execute the current line and move to the next line ignoring function calls
step	Step into functions called at the current line

These are the commands given on the PESU Academy Slides:

- **n** stands for 'execute next line'
- **p <variable>** prints variable value
- **l** lists the source code
- **b** to list all breakpoints
- **h** help
- **q** to quit the debugger
- **u** (up) goes up one level in the backtrace
- **d** (down) goes down one level in the backtrace
- Using **Breakpoint** we can tell the debugger the specific location to stop this can be achieved by 'break' command.
- Commands in breakpoint are:

Command	Function
break < line number >	Sets breakpoint on specified line number
break	Lists all breakpoints set
clear	To delete all breakpoints
break < functionName >	Sets breakpoint to the first line of a function
disable	to not halt execution when line is reached

How do you run pdb from the program itself rather than the command line?

Testing

```
graph TD; Planning --- Defining; Defining --- Designing; Designing --- Building; Building --- Testing; Testing --- Deployment; Deployment --- Planning; SDLC((SDLC))
```

- 

[illegible]

If you try to design something that's idiot-proof, the universe will design a better idiot

So don't worry about making it too idiot proof too

- Many tools are available to automate testing
- The frameworks that provide testing:
 - Robot
 - pytest
 - unittest
 - DocTest
 - Nose2
 - Testif
- We will be learning pytest since it's *free and open-source*. Google what that means if you're not aware, it's a deep rabbit hole to jump into. In short, it means that the software is fully free to use and anybody can view the source code of the project.

Pytest

Why Pytest?

- It allows multiple tests to run in parallel, which reduces the execution time of the test suite.
- It detects the test file and test function, if not mentioned explicitly.
- It allows you to skip the tests during execution.
- It allows the user to run a subset of the entire test suite.
- It is free and open source (FOSS)
- It has a simple syntax and is good for beginners.

Installation

You can use pip to install pytest - just run this in a terminal/command prompt:

```
pip install pytest
```

SHELL

- You can confirm the installation by running

```
pytest -h
```

SHELL

Identifying test files and functions

- You have to write python code to test your python code

Files:

- All your python files that you intend to use as a test must either *start* or *end* with the word 'test' along with an underscore.
- So it has to either be *test_someFileName.py* or *someFileName_test.py*

file name	is valid
test_one.py	valid
one_test.py	valid
testTwo.py	not valid
twoTest.py	not valid

- Pytest will automatically detect these files as 'test' scripts - aka the programs you write to test your other programs.
- **Important:** even if your file name does not follow the above rules, you can ask pytest to consider your files as test files - overriding the default rules. You can do this by running the python file as:

```
pytest testMethod.py
```

SHELL

- **HOWEVER**, This is *not possible* for the rules that govern function names - check out the next section:

Functions:

- Your test functions **should always start** with *test*
- There is no need of underscore - unlike file names.

function name	is valid
def test_method():	valid
def testMethod():	valid
def methodTest():	not valid

The `assert` keyword

In Python, the `assert` keyword is a debugging aid that tests a condition. If the condition is `True`, then the program continues to execute as normal. If the condition is `False`, then the program will raise an `AssertionError`

Here's an example of how to use the `assert` keyword:

```
x = 5 assert x > 0, "x must be positive"
```

PYTHON

In this example, the condition `x > 0` is `True`, so the program will continue to execute without raising an exception.

On the other hand, if we change the value of `x` to be negative, then the condition `x > 0` will be `False` and the program will raise an `AssertionError`:

PYTHON

```
x = -5 assert x > 0, "x must be positive"
```

This would cause the following exception to be raised:

PYTHON

```
AssertionError: x must be positive
```

The `assert` keyword is often used as a quick and easy way to check for conditions that should never occur in the program, and to raise an exception if one of those conditions is encountered. It can be a useful tool for debugging and testing your code.

So how do you write test files now?

PYTHON

```
# Contents of test.py:

def squareRoot(x):
    return x**0.5

def test_sqrt():
    num = 25
    assert squareRoot(num) == 5 # correct
```

- You have to run this program with the command `pytest test.py` instead of the usual `python test.py`. This will make the test functions run inside `test.py`.

The output on running the above program with pytest would be:

SHELL

```
===== test session starts =====
platform darwin -- Python 3.10.9, pytest-7.2.0, pluggy-1.0.0
rootdir: /Users/anuragrao
collected 1 item

test.py . [100%]

===== 1 passed in 0.00s =====
```

Now if you make the `assert` statement to return False somehow, i.e., the test fails:

PYTHON

```
def test_sqrt():
    num = 25
    assert squareRoot(num) == 6 # incorrect
```

`pytest test.py` would produce:

```
===== test session starts =====
platform darwin -- Python 3.10.9, pytest-7.2.0, pluggy-1.0.0
rootdir: /Users/anuragrao
collected 1 item
```

```
test.py F [100%]
```

```
===== FAILURES =====
----- test_sqrt -----
```

```
def test_sqrt():
    num = 25
>    assert squareRoot(num) == 6
E      assert 5.0 == 6
E      + where 5.0 = squareRoot(25)
```

```
test.py:8: AssertionError
```

```
===== short test summary info =====
FAILED test.py::test_sqrt - assert 5.0 == 6
===== 1 failed in 0.04s =====
```

- You can write test conditions like these where you use an assert statement with your function you want to test, a mock input and comparing it against the expected output. You can put these test functions in a separate file altogether - but remember you have to follow the pytest naming conventions.

Iterators

- This topic was taught out of nowhere and is not related to the previous topics. Why? idk-
- Iterators are neatly implemented within for loops, comprehensions, generators, etc., but concealed from view. **In simple words, an iterator is just an object that can be iterated on.**
- A Python **iterator object** must implement two specific methods, **iter()** or **__iter__()** and **next()** or **__next__()**, which are referred to collectively as the **iterator protocol**.

Suppose you have a program like this:

```
list1 = [25, 78, 'coding', 'is', '<3']
# How do you iterate through this list without using a for loop?
```

iter()

The **iter()** function in Python returns an iterator for the supplied object. The **iter()** generates a thing that can be iterated one element at a time. These items are handy when combined with loops such as for loops and while loops.

Syntax:

```
iter( object , sentinel )
```

`iter()` function takes two parameters:

- Object: An object whose iterator needs to be created (lists, sets, tuples, etc.).
- Sentinel (optional): Special value that represents the end of the sequence.

next()

The `next()` function returns the next item from the iterator. The `next()` function holds the value one at a time.

Syntax:

```
next( iterator , default )
```

PYTHON

The `next()` method accepts two parameters:

- **Iterator:** `next()` function retrieves the next item from the iterator.
- **default(optional):** this value is returned if the iterator is exhausted (not tired, but no next item to retrieve).

Coming back to the question we had at the beginning - Iterating without a for loop:

```
X = [25, 78, 'Coding', 'is', '<3']
# Get an iterator using iter()
a = iter(X)

# Printing the a iterator
print(a)

# next() for fetching the 1st element in the list that is 25
print(next(a))

# Fetch the 2nd element in the list that is 78
print(next(a))

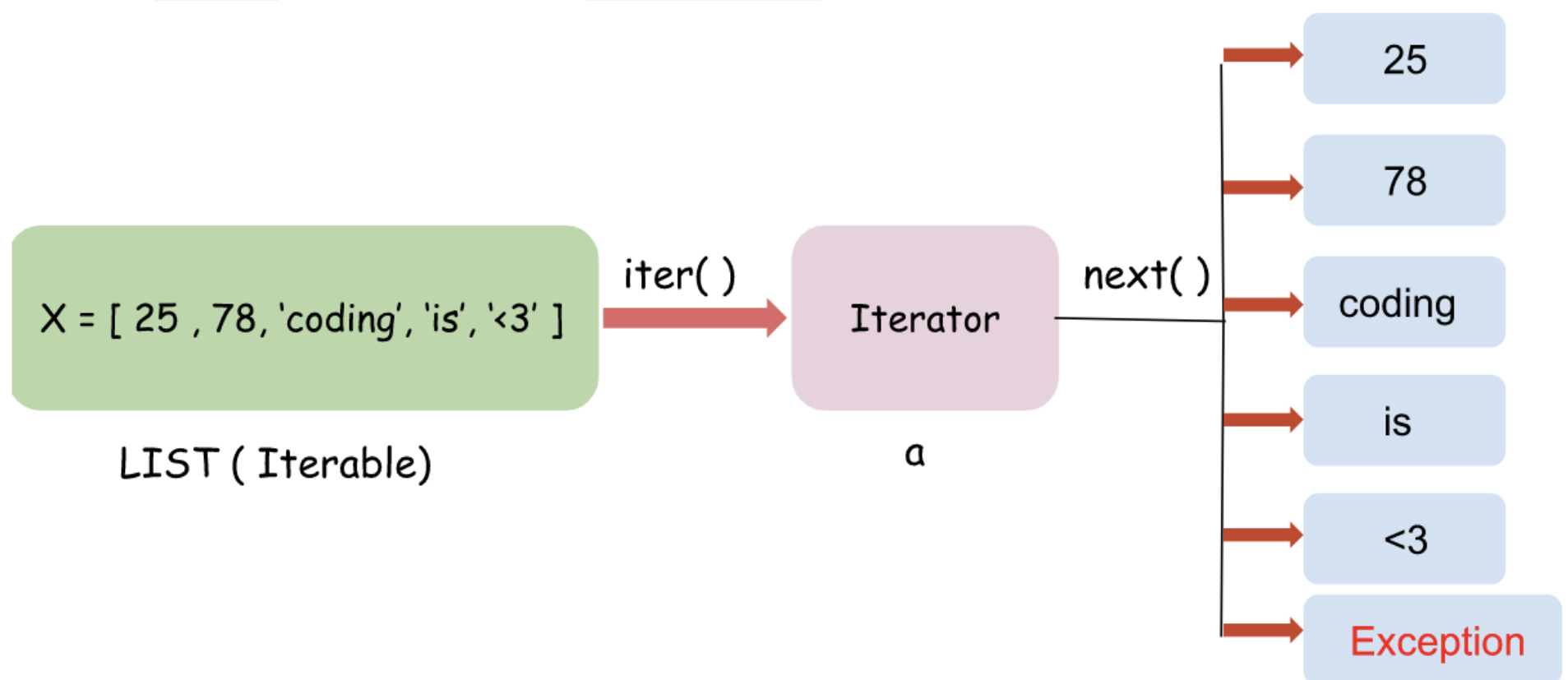
# Fetching the consecutive elements
print(next(a))
print(next(a))
print(next(a))
```

PYTHON

Output:

```
<list_iterator object at 0x000001B91F9CFFD0>
25
78
Coding
is
<3
```


Now if you `print(next(a))` again, the iterator is exhausted, There are no elements left on the iterator. If you try to `next(a)` again, it'll throw a `StopIteration` error.



- You cannot `next(a)` for every element, so you can use a while loop for it along with a `try` and `except` block. If you don't know what this is, don't worry, it's not in syllabus :)

```
# Program to print the tuple using Iterator protocols
tup = (87, 90, 100, 500)

# get an iterator using iter()
tup_iter = iter(tup)

# Infinite loop
while True:
    try:
        # To fetch the next element
        print(next(tup_iter))
        # if exception is raised, break from the loop
    except StopIteration:
        break
```

PYTHON

Output:

```
87
90
100
500
```

PYTHON

- This is how for loops are implemented under the hood of Python.

Important

You can use a for loop to iterate through an iterator too. BUT! This weird thing happens if you try to loop through an iterator twice:

```
>>> l = [1,2,3,4]
>>> a = iter(l)
>>> for i in a: print(i)
...
1
2
3
4
>>> # SECOND TIME:
>>> for i in a: print(i)
...
>>> # DOES NOT PRINT ANYTHING!
```

- This happens because the first time the for loop runs, the pointer goes through the iterator and ends up at the end of the original list.
- Now when we run the for loop once more, it doesn't print anything *because the pointer of the iterator is already at the end of list, so no more elements are left*
- I am 99% sure this shit will come in ISA to trick us, beware!

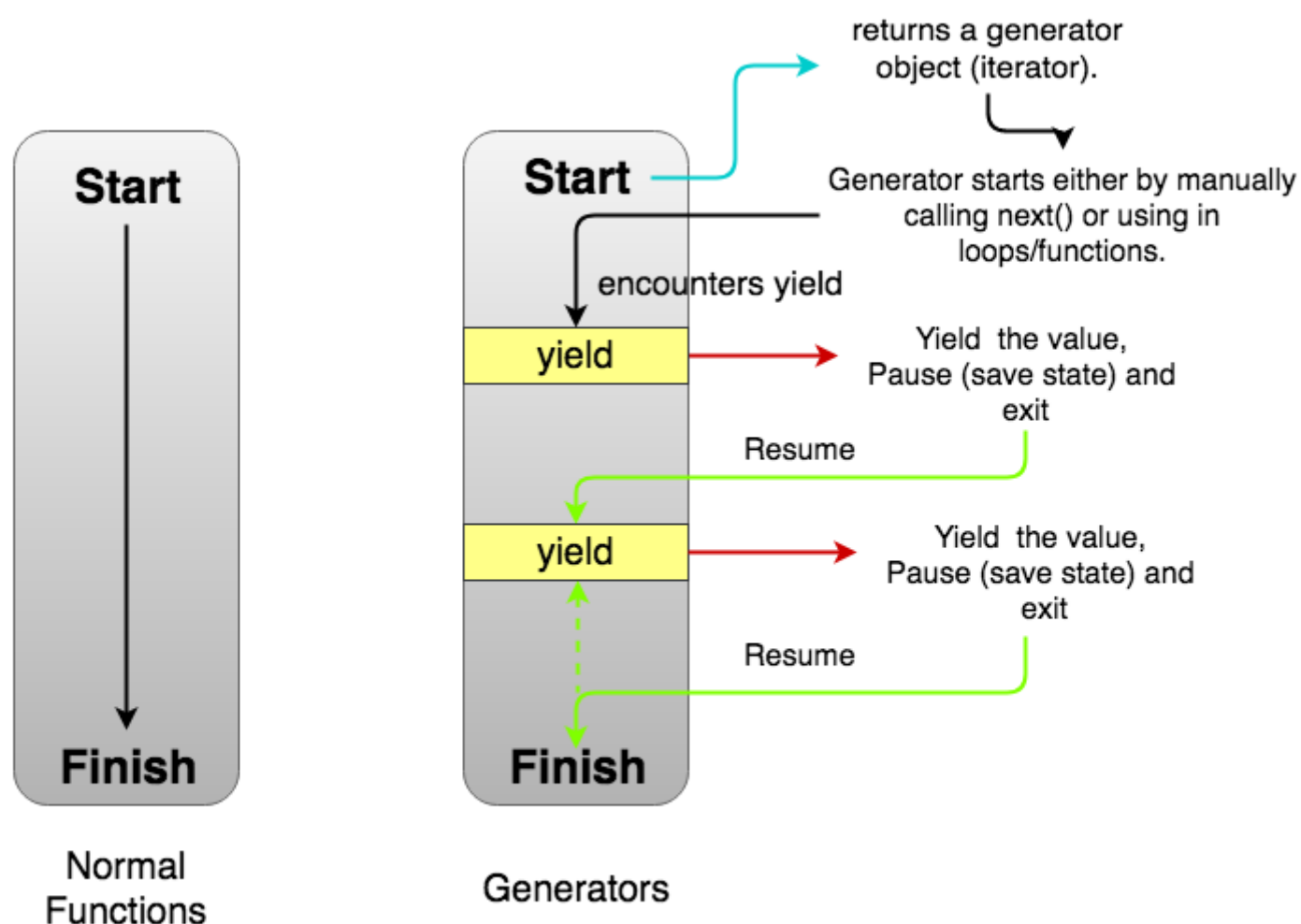
Generators

Building an iterator in Python requires a significant amount of effort. We must create a class containing `__iter__()` and `__next__()` methods, keep track of internal states and raise `StopIteration` when no values are returned. This is both long and contradictory. In such cases, the *generator* comes to the rescue.

Python has a generator that allows you to create your iterator function. A generator is somewhat of a function that returns an iterator object with a succession of values rather than a single item.

A *yield* statement, rather than a return statement, is used in a generator function.

The difference is that, although a return statement terminates a function completely, a yield statement pauses the function while storing all of its states and then continues from there on subsequent calls.



Let's take an example, Imagine you need to create a function that takes a `max_element` as a parameter and then return an `iterator` of squares of numbers until the `max_element`. You can implement it in this way:

```
def getSquares(max_element):  
    for i in range(1, max_element+1):  
        yield i**2 # adds i^2 to iterator  
  
print(getSquares(7))  
for i in getSquares(7): print(i)
```

PYTHON

Output:

```
<generator object getSquares at 0x101003ca0>  
1  
4  
9  
16  
25  
36  
49
```

SHELL

Why use generators when the return statement is already present?

Generators are really memory efficient. A standard function that returns a sequence will first construct the whole sequence in memory before returning the result. If the number of elements in the sequence is enormous, this is overkill. The generator implementation of such sequences, on the other hand, is memory-friendly and preferable since it only generates one item at a time.

Important

```
# A simple generator for Fibonacci Numbers
def fib(max):
    # Initialize first two Fibonacci Numbers
    p, q = 0, 1

    # yield next Fibonacci Number one at a time
    while p < max:
        yield p
        p, q = q, p + q

# Ask the user to enter the maximum number
n = int(input("Enter the number up to which you wish the Fibonacci series to be
printed: \n"))

# Create a generator object
x = fib(n)
# Iterating over the generator object using for
# in a loop.
print("Resultant Series up to", n, "is :")
for i in x:
    print(i)
```

Output:

SHELL

```
Enter the number up to which you wish the Fibonacci series to be printed:
5
Resultant Series up to 5 is :
0
1
1
2
3
```

- **yield** in Python can be used as the **return** statement in a function. The function returns a generator that can be iterated upon instead of returning the output when done so.
- Python iterates over the code until it reaches a **yield** line within the function. The function then transmits the produced value and pauses in that state without leaving. Then the loop continues executing.

Revision Questions:

- Q27. Convert all strings in a list to uppercase
 Q28. If string length > 3, return uppercase
 Q29. Add each element of a list
 Q30. Odd numbers
 Q31. difference between 2 lists using filter

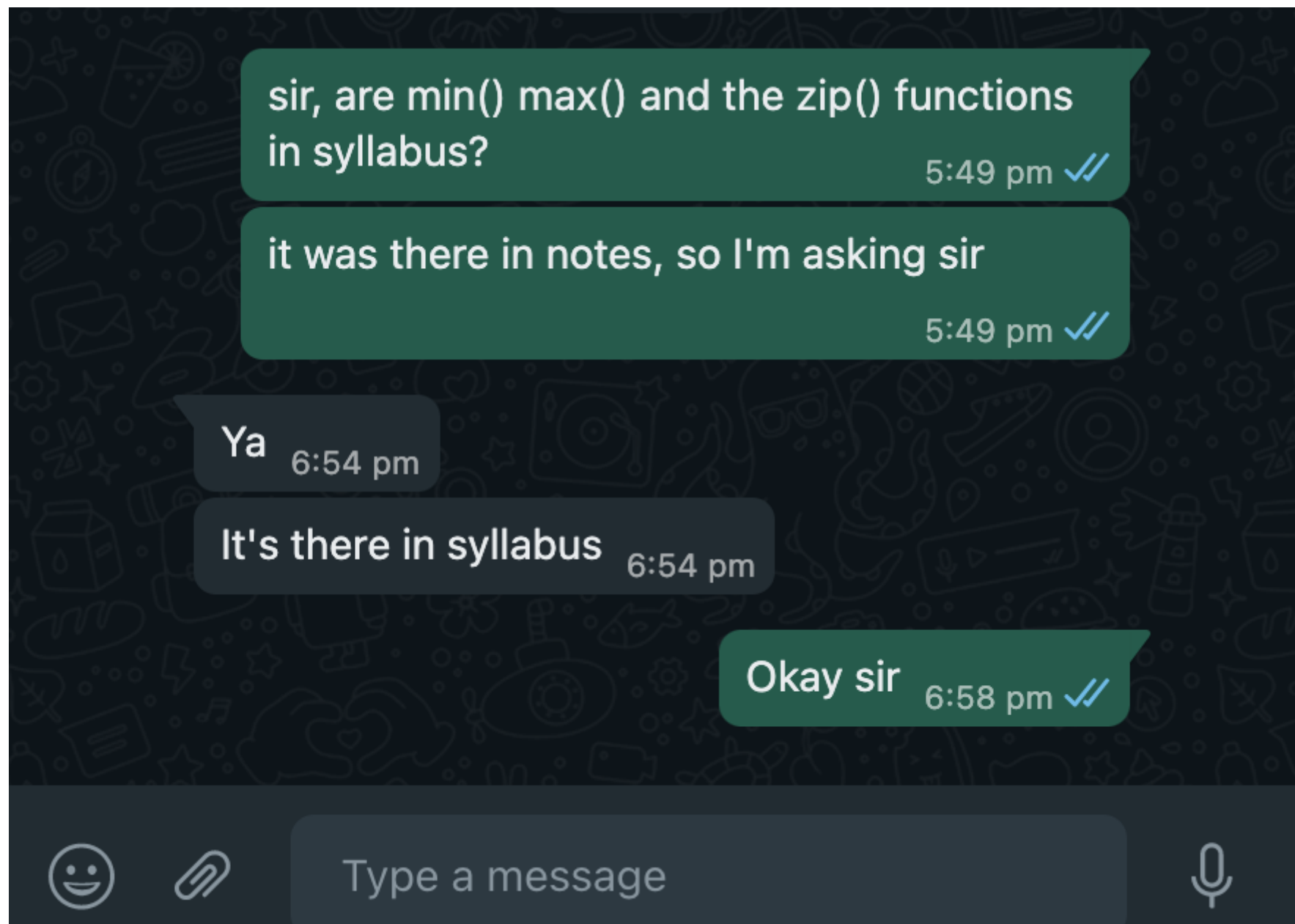
Q32. remove stop words in a string using filter()

Q33. Using the filter function, define a filterPositive function that takes a list of numbers and returns a list of its positive elements.

Q34. Write a python program to find the maximum of a list of numerical values by using reduce() function

Q35. Find factorial using reduce.

Stuff that was not taught but then later told it is there in syllabus:



The `min()` function

- It returns the smallest item in an iterable object
- Basically, find the minimum
- It can also be operated on a list of strings - in that case it returns the lexicographically smallest string
- syntax: `min(<iterable>, <optional key>, <optional default>)`
- Note that instead of an iterable, you can also just pass separate integers and it will return the minimum of them

The `key` parameter

- The `<key>` parameter mentioned above takes a function, similar to how map takes a function for its 'callable' field.
- It is a named argument, so you have to mention `key=someFunction()`
- It basically defined the key to find the minimum by. This is explained better with an example:

```
# Using an iterable
numbers = [5, 2, 9, 1, 7]
smallest_number = min(numbers)
print(smallest_number) # Output: 1

# Using two or more separate arguments
a = 5
b = 9
c = 2
smallest_number = min(a, b, c)
print(smallest_number) # Output: 2

# Using key
numbers = [5, 2, 9, 1, 7]
smallest_number = min(numbers, key=lambda x: x**2)
print(smallest_number) # Output: 1
# here the list is sorted by the squares of each numbers instead of the numbers
itself and the min element is returned. In this case, it still returns 1 because
even with a square, it is still the smallest number.
```

- The `max()` function is the exactly the same as the `min()` function but it returns the max element instead.
- The `min()` and `max()` functions also have a `default` parameter, which allows you to specify a default value to be returned if the iterable passed to the function is empty:

```
numbers = []
smallest_number = min(numbers, default=0)
print(smallest_number) # Output: 0
```

The `sorted()` function

- This function, as the name suggests is used to sort a list of items.
- The syntax is like this:

```
sorted(iterable, key=None, reverse=False)
```

- The iterable is what you want sorted. Default is ascending order.
- The `key` works similar to the `key` in `min()` and `max()`.
- If reverse is set to True `reverse=True`, then it will return the elements in descending order.
- The `key` and `reverse` are optional parameters. The default value of them is mentioned in the syntax.
- Example:

```
numbers = [3, 1, 4, 2, 5]
sorted_numbers = sorted(numbers)
```



```
print(sorted_numbers)
# Output: [1, 2, 3, 4, 5]
```

- With **key**:

```
items = [("apple", 3), ("banana", 2), ("orange", 1)]
sorted_items = sorted(items, key=lambda x: x[1])
print(sorted_items)
# Output: [("orange", 1), ("banana", 2), ("apple", 3)]
```

PYTHON

In the above example, the lambda function returns the element at the index[1] of each element. Since our list has elements that are tuples, the index[1] of this tuple is taken as the key and is sorted by that.

- With **reverse**:

```
sorted_items = sorted(items, key=lambda x: x[1], reverse=True)
print(sorted_items)
# Output: [("apple", 3), ("banana", 2), ("orange", 1)]
```

PYTHON

The **zip()** function

- It combined two or more lists.
- How? it creates another list with tuples in them. The first element of this tuple comes from the first element of the first list, and the second element of the tuple from the second list.
- Note that if the input iterables are of different lengths, the iterator stops generating tuples as soon as the shortest iterable is exhausted.
- It returns an iterable object. That means if you directly try to print it, it will print something similar to **<zip object at 0x...>**
- Example:

```
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]

print(list(zip(names, ages)))
# Output: [('Alice', 25), ('Bob', 30), ('Charlie', 35)]
```

PYTHON

- Example where the length of last name is lesser than the first name. So the iterables are exhausted:

```
first_name = ['Alice', 'Bob', 'Charlie']
last_name = ['Smith']

print(list(zip(first_name, last_name)))
# Output: [('Alice', 'Smith')]
```

PYTHON

Unzipping


- You can unzip a zipped object by passing it to the zip function but with a *

```
zippedList = [('Alice', 25), ('Bob', 30), ('Charlie', 35)]
name, age = zip(*zippedList)
print(name)
print(age)
```

PYTHON

```
# output
# ['Alice', 'Bob', 'Charlie']
# [25, 30, 35]
```

Find all the programs on my  [GitHub](#)

Made with  by Anurag Rao