

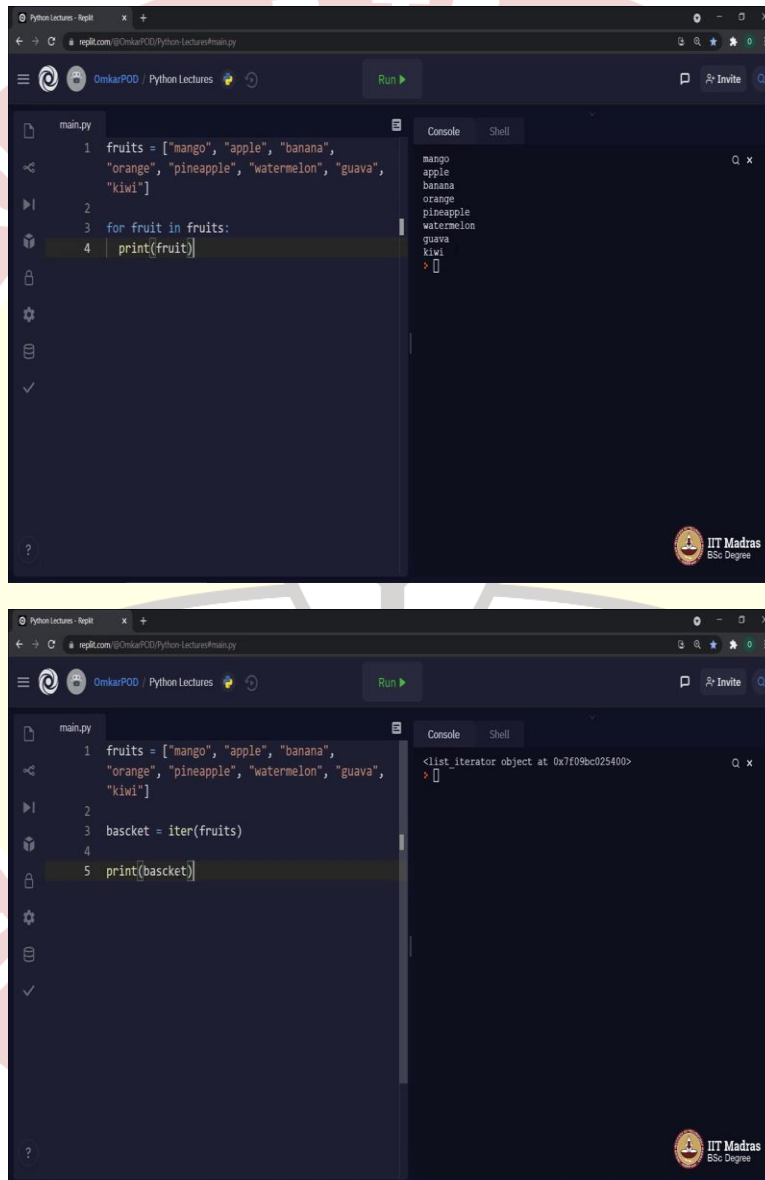


IIT Madras

ONLINE DEGREE

Programming in Python
Professor Sudarshan Iyengar
Department of Computer Science & Engineering
Indian Institute of Technology, Ropar
Functional Programming (Part 1)

(Refer Slide Time: 00:16)



The image displays two screenshots of a Python IDE interface, likely Replit, showing the execution of Python code. The background features a large, semi-transparent watermark of the Indian Institute of Technology Madras logo.

Top Screenshot: The code in `main.py` defines a list `fruits` and iterates over it using a `for` loop.

```
1 fruits = ["mango", "apple", "banana",  
2 "orange", "pineapple", "watermelon", "guava",  
3 "kiwi"]  
4 for fruit in fruits:  
5     print(fruit)
```

The console output shows the fruits listed one by one:

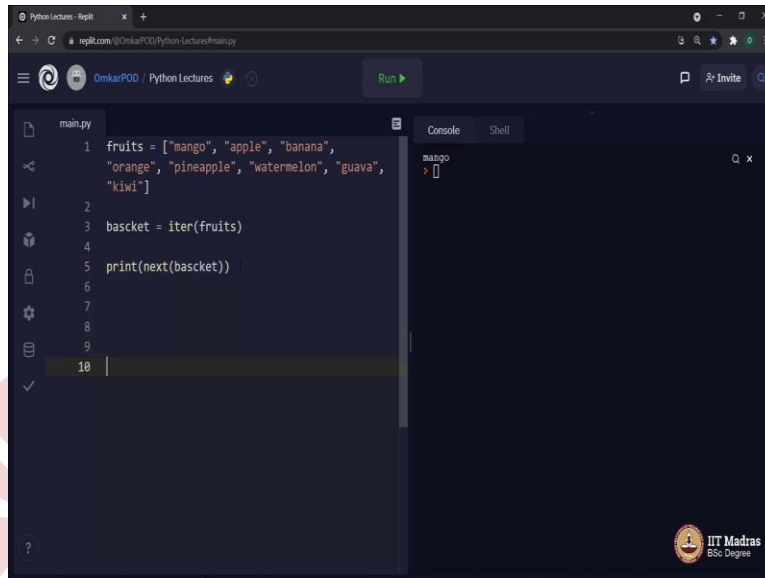
```
mango  
apple  
banana  
orange  
pineapple  
watermelon  
guava  
kiwi  
> []
```

Bottom Screenshot: The code in `main.py` defines the same list `fruits` and creates an iterator object `basket` using the `iter()` function.

```
1 fruits = ["mango", "apple", "banana",  
2 "orange", "pineapple", "watermelon", "guava",  
3 "kiwi"]  
4 basket = iter(fruits)  
5 print(basket)
```

The console output shows the iterator object:

```
<list_iterator object at 0x7f09bc025400>  
> []
```

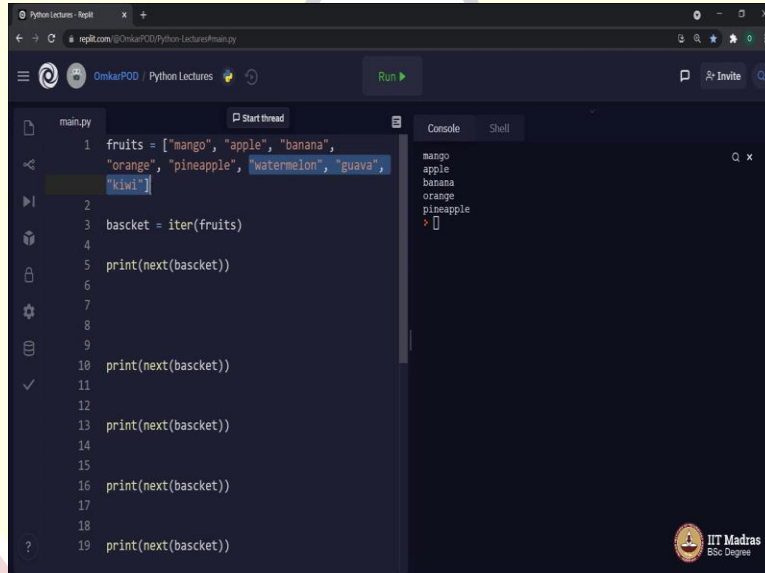


The screenshot shows a Python REPL window with a file named `main.py`. The code defines a list of fruits and an iterator. The console shows the first element of the iterator.

```
1 fruits = ["mango", "apple", "banana",  
2         "orange", "pineapple", "watermelon", "guava",  
3         "kiwi"]  
4  
5 basket = iter(fruits)  
6  
7 print(next(basket))  
8  
9  
10
```

Console output:

```
mango  
>
```



The screenshot shows the same Python REPL window, but now with a `Start thread` button and more code. The console shows the first five elements of the iterator.

```
1 fruits = ["mango", "apple", "banana",  
2         "orange", "pineapple", "watermelon", "guava",  
3         "kiwi"]  
4  
5 basket = iter(fruits)  
6  
7 print(next(basket))  
8  
9  
10 print(next(basket))  
11  
12 print(next(basket))  
13  
14  
15 print(next(basket))  
16  
17  
18 print(next(basket))  
19
```

Console output:

```
mango  
apple  
banana  
orange  
pineapple  
>
```

Hello Python students. In this lecture, we will talk about some concepts which comes under something called as functional programming in Python. Let us look at this particular code block. There is a list of strings. Each string is a name of a fruit. And I am iterating over that list using a regular for loop and printing each fruit in every iteration. And if I execute it, I will get a list of all these fruits one after other in every new line. This is something which we saw at the very beginning of this particular course.

But now consider a real time situation where I have a basket full of fruits and I am trying to distribute all these fruits among all the kids in my neighborhood. So, I have that basket with me. And I am just walking around in my neighborhood and whenever I come across a kid I am

picking one fruit from my basket and giving it to that particular kid. And this is how we do things in real time when we want to distribute maybe fruits, chocolates or anything.

But when we try to implement that particular thing as it is in Python program, our code will look something like this, where we have a list representing all these fruits, which is like a basket, and then we will iterate over that list using for loop and print name of each fruit, as in this is more like a handing over that fruit to a kid.

But if you observe here, after printing that mango, the apple also gets printed immediately. But whereas, in real time, that may not happen. I met one kid, I may not come across next kid immediately. There might be some gap in between where all the remaining fruits are still in my basket and I am simply walking around, maybe doing something else. But these loops do not have that flexibility. It stops only when the entire list is exhausted, as in there is no new element remaining in the list. That is the place our loop stops. We do not have any choice on when to pick the fruit from particular list.

So, in order to solve this problem, Python has a specific feature called iterator, which is very similar to this iterative process or these iterations which we do with these loops. But over there, we have a control over that iterator. So, let me modify this code in such a way to accommodate that real time example which I just explained.

Let us say, I have a basket, just a variable and over here I will call one function called iter and I will pass this particular list called fruits as a parameter to this particular function. And Python will take this particular fruits list as a parameter and it will convert this into an iterator which means now this basket is a iterator.

Let us print first and see what basket is. It says basket is a list iterator object, because originally fruits was a list and then it became iterator. So, collectively, it became list iterator object. And now this iterator named basket will work exactly the way a basket full of fruits works in real world. I will say next of basket and execute the code. It will say mango.

Now, in between, I have something else to do, just like in the real world, I am walking around doing something else, but not distributing any fruits because I have not come across any kid so far. Then I see one kid walking towards me. So, I call this again, as in I put my hand in my

basket and get next fruit. And if I execute this, it will give me next fruit. Similarly, as many number of times I execute it, every time I will get next fruit. Still, if you see, I executed this for five times, that is why I got only five fruits. But still these remaining three fruits are still there in my basket.

If I come across three more kids, I will distribute these fruits or else I may end up not distributing these fruits and those fruits will remain as it is in my basket, just like how it will happen in real world. And this is what this particular concept of iterator is. The concept of iterator involves two different functions.

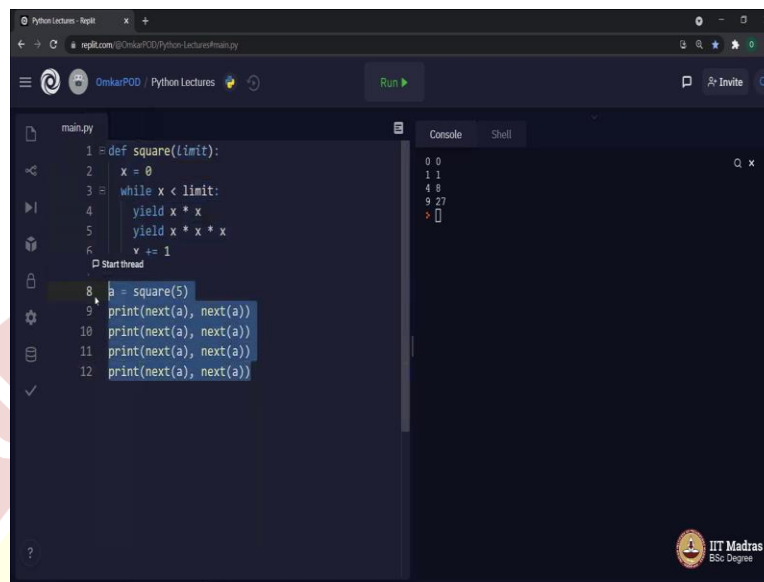
One is called iter, which converts any iterable entity, let me repeat, any iterable entity, as in list is iterable. You can iterate over a list. You can iterate over a string. You can iterate over a tuple and so on. So, anything on which your regular iterator, your regular loop works, all those entities are referred as iterable. So, you can use any iterable entity and convert it into an iterator of that particular type as we saw list iterator, we can create string iterator and so on.

Then this is our, that iterator. After that using this next function we can iterate over that particular iterator if and when required. And this is what we have seen earlier with respect to file handling as well. And that was the difference between read line and read lines. Whenever we called read lines, we were able to read all the lines from the file. But whenever we say read line, it was reading only one line at a time. When we were calling read line function first time it was reading first line of the file. When we called it second time it read second line from the file and so on.

Similar thing happened with that f dot seek function as well. Internally all these functions, even a regular for loop, where we say for fruit in fruits, internally, it do the same thing whatever we are doing right now. But over there, we do not have control on this particular iterator. Whereas in this case, we will control this iterator using this function called next. This is regarding iterator.

But what if I do not want to create iterator from a list, which means if I am supposed to iterate over such a basket, why do I have to create this list in a first place? Why cannot I put these fruits directly into basket and iterate over it where I can generate this particular iterator on my own without even creating this list. And in order to answer that Python has a solution for that as well. And it is referred as generator. We can generate our own iterator.

(Refer Slide Time: 09:03)



The screenshot shows a Python IDE with a file named 'main.py'. The code defines a generator function 'square(limit)' that yields 'x * x' and 'x * x * x' in a while loop. The function is called 'a = square(5)', and its next values are printed using 'print(next(a))'. The console output shows the sequence of values: 0 0, 1 1, 4 8, and 9 27.

```
1 def square(limit):  
2     x = 0  
3     while x < limit:  
4         yield x * x  
5         yield x * x * x  
6         x += 1  
7  
8 a = square(5)  
9 print(next(a), next(a))  
10 print(next(a), next(a))  
11 print(next(a), next(a))  
12 print(next(a), next(a))
```

Console Output:

```
0 0  
1 1  
4 8  
9 27  
>
```

So, now let us clear the screen and write a Python program to create generator, def square and parameter is limit, x equal to 0, while x is less than limit. This yield is a new keyword. Usually along with any function we use return. But whenever we try to create a generator, we do not have any written statement. We have a yield statement. So, what happens is, after this, we will call this particular generator the same way we call any other function by saying a is equal to square and some limit value over here. Let us first execute this then we will see what that yield is all about.

The program is executing without any error. And as we are not printing anything, we are not getting any output. After this, what I will do is, I will simply say print and I will use that next function which we saw just now on this variable a. Next a and once again next a. Let us first execute, then I will come to explanation. It says 0, 0. If I repeat this, it says 0, 0, 1, 1. Let me repeat one more time. Now, it is 4, 8. What is happening over here is, whenever we say a is equal to square with limit 5, this particular function gets called. And this function is referred as generator, because instead of written statement, we have a yield statement.

So, what happens is, we come to while this condition is true. So, in such a case, this particular function, this particular generator yields its first output which is x into x. That is how we got this first 0. Then, in our next yield statement, we said x into x into x, this is x cube. So, we have got this second 0. And this will keep happening until x is less than limit as we have set limit to 5. So,

this will happen till x becomes 5. So, from 0 to 4, this will happen. When x is 0, this is the output. When x is 1, this is the output. When x is 2, this is the output.

But as you can see, once again, this generator generates an iterator. So, a is an iterator and we can iterate over that iterator using same next function which we just saw. Even though a already holds all those values, it is us who now have a control on how many times we want to actually use it. So, this is how a combination of generator and iterator works together. Thank you for watching this lecture. Happy learning.

