CS521 O2
Information Structures with Python

Lecture 10

Guanglan Zhang

guanglan@bu.edu

Some slides adapted from Prof. Eugene Pinsky

Table of Content

- Function Annotations
- Iterators & Generators
- Others (lambda, recursion, main())
- Key takeaways

Function Annotations

- Annotations were introduced in Python 3.0
- For functions, we can annotate arguments and the return value
- The function writer can use annotations as suggestions to the function users
- Python does nothing with the annotation
- The annotations are stored as a dictionary at function definition time
- It becomes part of the function object, stored in the special variable __annotations__

```
def func(arg: arg_type, optarg: arg_type = default) -> return_type:
```

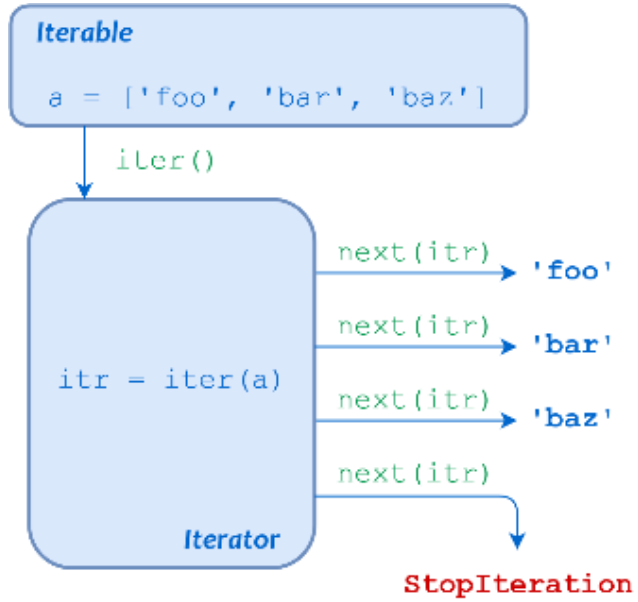
Iterators

- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values
- Iterable objects, such as string, list, tuple, dictionary, and set, are not iterator
- They are iterable containers that you can get an iterator from
- An iterator is a value producer that yields successive values from its associated iterable object
- Iterable objects have *iter*() method which is used to get an iterator
- Iterator objects support two attributes, __*iter*__() and __*next*__() methods
  - ✓ *next*(iterator) – takes in an iterator, returns the next element in the iterator
  - ✓ *iter*(iterable) – returns an iterator

Python *for* Loop

```
for <var> in <iterable>:
    <statement(s)>
```

- A *for* loop is used to iterate through an iterable
- The *for* loop creates an iterator
- In each loop, *next*() is executed and returns the result to the loop variable <var>
- Until *next*() call results in StopIteration exception
- The *for* loop captures the exception and end the loop



Generators and *yield*

- Generator functions are a special kind of function that return an iterator
- Generator functions use *yield* instead of *return*
- When *yield* expression is evaluated, the function generates its next value, only its next value, return the value to the caller
- Unlike return, it does not exit the function afterward
- The state of the function is “remembered”
- When *next*() is called on a generator object, the next value is generated
- When the function ends (cannot yield any more data), *next*() raises a StopIteration exception

Generators and *yield*

- Generator functions are a special kind of function that return an iterator
- Generator functions use *yield* instead of *return*
- When *yield* expression is evaluated, the function generates its next value, only its next value, return the value to the caller
- Unlike return, it does not exit the function afterward
- The state of the function is “remembered”
- When *next*() is called on a generator object, the next value is generated
- When the function ends (cannot yield any more data), *next*() raises a StopIteration exception

Lambda function

- It is a small anonymous function. It takes any number of arguments, but can only have one expression
- It is often used as an argument to a higher-order function (a function that takes in other functions as arguments), such as *filter(), map()* and *reduce()*
- *map()* – takes a function and one or more iterables as input arguments, it applies the function to each of the element in the iterable
- *Filter()* - – takes a Boolean function and one iterables as input arguments, it applies the function to each of the element in the iterable and only collect those elements for which the function returns True
- *reduce()* – part of *functools* module. Takes a function and one iterables as input arguments. It applies the function to the 1st two elements; then it reapplies the function to the previous result and the 3rd element, and so on.

Recursion: a control mechanism

- Recursive function is a function calls itself
- Recursive: more execution time
- Non-recursive: typically large code

Factorial function n! = 1 * 2 * 3 …(n-1) * n

Recursive computation
0! = 1
1! = 1
2! = 2
3! = 1 * 2 * 3 = 2! * 3 = 6
4! = 1 * 2 * 3 * 4 = 3! * 4 = 24

n! = (n-1)! * n

main() function

- Put code that takes a long time to run or has other effects on the computer in a function or class
- When Python sees *def* or *class* keywords, it stores the definition, but does not execute them until you tell it to
- Use the *if __name__ == "__main__"* idiom to conditionally run main() only when __name__ is equal to "__main__“
- When the __name__ variable equal to "__main__“, it means the Python interpreter is executing your script and not importing it as a module
- Python does not assign any significance to the name main(), still the best practice is to name the entry point function main()
- main() contains code that you want to run when Python executes the file

Exercises

1. An arithmetic progression A(a, b) is a sequence of numbers, where $x_1 = a$, $x_2 = x_1 + b = a + b$, ..., $x_n = x_{n-1} + b = a + (n - 1)*b$

   Write a generator function gf(a, b) to produce the next value in A(a, b)

2. Write a recursive function that returns the n^{th} Fibonacci number given a user input integer n. (The Fibonacci Sequence is the series of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... The next number is the sum of the two numbers before it.)

Key takeaways

- An iterator is a value producer that yields successive values from its associated iterable object
- Iterator objects support two attributes, __*iter*__() and __*next*__() methods
- Generator functions returns an iterator using *yield*
- When *yield* expression is evaluated, the function generates its next value, only its next value, return the value to the caller
- Unlike return, it does not exit the function afterward
- The state of the function is “remembered”
- When *next*() is called on a generator object, the next value is generated
- When the function ends (cannot yield any more data), *next*() raises a StopIteration exception