Lecture 13 - Some More Function Topics

Computer Programming

Robert D. Vincent and Samia Hilal

Marianopolis College

April 10, 2022

Advanced function topics

- `enumerate()`
- Default argument values.
- Keyword arguments.

Using for Loop to Iterate a List

- There are different methods to iterate over a list.
- Each method has a special purpose.
- We will consider 3 different methods.
    1. Iterate over items
    2. Using range to iterate over indices
    3. Using enumerate function to iterate over both items and indices
- The next 5 slides give a review of methods that we have seen.

Iterating over list items

- **Syntax:**

```
L = [10, 50, 75, 83, 32]
for item in L:
```

- Iterate directly through the list
- Get access to one list item in each iteration
- Example: Print info for each list item:
  - complete list
  - count of each item
- Can be used to check if an item is on the list (item index not required)
  - Example: Find a student on the list and print info.
- Compute a sum or product of items on the list:
  - Sum of the squares of items on the list
  - The sum of integer values of list items
  - The product of all even integers on the list

Iterating over list items

- **Example:**

```
L = [10, 50, 75, 83, 32]
total=0
for item in L:
    total += item
    print(item)
```

- The temporary variable item gets the next list element value at each iteration but `does not have access to the list.`
- changing item does not change the list.
- What if we need to change the list? Use range method.

range to iterate over list indices

- The range() function returns a sequence of integers.
- Use range() to return the indices of the list.
- **Syntax:**

```
L=[10, 50, 50, 75, 75, 75, 83]
for i in range(len(L)):
    if (L[i]!=L[i-1]):
        print(L[i],"x",L.count(L[i]))
```

- range(len(L)) or range(7) in this example returns the values: 0,1,2,3,4,5,6
- These are the indices of all items in L.
- Access each list item directly using L[i]

When to iterate using range

- Iterating over list items is simple but cannot be used for all applications.
- range method can be used all the time.
- range method required in these cases:

1. When we need to update item(s) on the list:

```
L=['10','50','75','83']
for i in range(len(L)): #make items int
    L[i]=int(L[i])
```

2. When we need to compare list items:

```
L=[10, 50, 50, 75, 75, 75] #duplicates
for i in range(len(L)):
    if (L[i]!=L[i-1]):
        print(L[i],"x",L.count(L[i]))
```

When to iterate using range

3. When we need to return item index (position):

```
def find_item(lst,item):
    """returns the position of an item on
       or -1 if not found"""
    for i in range(len(lst)):
        if (lst[i] == item):
            print(L[i],"at position ",i)
            return(i)
    return(-1)

L=[10, 50, 50, 75, 75, 75, 83]
find_item(L,50)
```

The `enumerate()` function

- `zip()` can be used to iterate over *both* the values and indices of a list:

```
fr = ["apple", "grape", "peach"]
for i, v in zip(range(len(fr)), fr):
    print(i, v)
```

- This will print:

```
0 apple
1 grape
2 peach
```

- The function `enumerate()` simplifies this.

enumerate() examples

```
>>> x = [40, 45, 36]
>>> for ind, val in enumerate(x):
...     print(ind, val)
...
0 40
1 45
2 36
>>> y = ['n', 's', 'e', 'w']
>>> for ind, val in enumerate(y):
...   print(ind, val)
...
0 n
1 s
2 e
3 w
```

Default argument values

- Problem: Certain function arguments may not always be needed.
- Solution: Allow default values for "missing" function parameters.
- We do this by adding an equal sign and an expression after a parameter name:

```
def func1(name1, name2 = expression):
```

- If the second argument is not specified in a call to `func1()`, `name2` will be equal to *expression*.

Why is this useful?

- Suppose we want to write a `search()` function similar to `index()`.
- The normal case is to start searching from the beginning of a list, at index 0.
- However, what if we have repetitions, and want to see if there are many items on the list with the same value?

Revised search function

```
def search5(my_list, value, start = 0):
    '''Linear search 'my_list' for
    'value', starting at 'start'.'''
    for ind in range(start, len(my_list)):
        if my_list[ind] == value:
            return ind
    return -1
```

- The '= 0' after the parameter name `start` means that if only two arguments are passed to the function, `start` will be set to zero.

Revised search function

```
>>> from search5 import search5
>>> int_list = [8, 5, 9, 2, 7, 8, 1, 2]
>>> print(search5(int_list, 8))
0        # Search starts at zero.
>>> print(search5(int_list, 2))
3        # Finds first instance.
>>> print(search5(int_list, 8, 2))
5        # Finds second instance.
>>> print(search5(int_list, 2, 4))
7        # Finds second instance.
>>> print(search5(int_list, 5, 0))
1        # Ok to specify the value.
>>> print(search5(int_list, 5, 2))
-1       # No more 5's.
```

Another example

- Suppose we have a program that asks the user for an integer.
- We usually use the same message and limits:

```
def get_int(min_val = 1, max_val = 100,
            prompt = 'Type a number:'):
   while True:
       r = int(input(prompt))
       if r >= min_val and r <= max_val:
           return r
       print('Enter a number between',
             min_val, 'and', max_val)

print(get_int())
```

Important details

- Any expression act as the default value.
- Once you specify a default argument for *any* parameter, *every* subsequent parameter must include a default argument.

```
def my_fn1(a, b = 2, c = 3): # OK
    return (a + b) / c

def my_fn2(a = 1, b, c = 3): # Illegal!
    return (a + b) / c
```

Another important detail

- The expression is evaluated only once, when the function is defined.
- Therefore, changes to any named value used in the expression will have no effect later on.

```
i = 4
def f(a = i):
  return a * 4

i = 5
print(f(4)) # Will print 16!
```

Final important warning

- Be careful with mutable values as default parameters.
- The default expression is evaluated only when the function is defined.
- One mutable object is used for the default value.

```
def f(a, L = []):
    L.append(a)
    return L

print(f('x')) # Prints ['x']
print(f('y')) # Prints ['x', 'y']
print(f('z')) # Prints ['x', 'y', 'z']
```

Keyword arguments

- We can assign a value to a specific parameter by using a *keyword argument* of the form:

```
  name = expression
```

- Applies for function *calls* rather than definitions
- This works with *any* function, whether or not it uses default argument values.
- Overrides the normal positional assignment of values to parameter names.
- You have already seen an example: the `end` argument to `print()`.

Keyword argument, example 1

```
def f(a, b):
    return a + b

print(f("X", "Y")) # Prints XY
print(f(a = "X", b = "Y")) # Prints XY
print(f(b = "X", a = "Y")) # Prints YX (!)
print(f("X", b = "Y")) # Prints XY
```

But once you specify a keyword argument, all subsequent arguments must be keyword arguments:

```
def f(a, b):
    return a + b

print(f(a = "X", "Y")) # Syntax error
```

Keyword argument details

- Assigns an argument to a specific parameter.
- The parameter name must appear in the function definition.
- It is *not* the same as an actual assignment!

```
def f(a, b):
  return a / b

print(f(1, 2)) # Prints 0.5
print(f(b=1, a=2)) # Prints 2.0
print(a, b) # ERROR: a and b aren't globals!
print(f(1, a=2)) # ERROR: 2 'a' and no 'b'!
print(f(1, c=2)) # ERROR: what's 'c'?
```