PYTHON LAB BOOK

Python For Programmers
UCSC Extension Online

Lab 6 Sequences

Topics

- Sequence types: `str`, `tuple`, `list`
- Sequence slicing and other manipulations

lab05_1.py

```
 1 #!/usr/bin/env python
 2 """ lab05_1.py
 3 1.  Write a function that returns a total cost from the sales price
 4 and sales tax.  The default value for the tax rate should be 8.25%.
 5 Test your function.
 6 """
 7 from __future__ import division
 8
 9 def Cost(price, tax=.0825):
10     return price * (1 + tax)
11
12 print " price |   .0825 |   .0925"
13 for price in range(50, 1001, 50):
14     dollars = price/100
15     print "$%5.2f | $%6.2f | $%6.2f" % (dollars, Cost(dollars),
16                                         Cost(tax=.0925, price=dollars))
17 """
18 $ lab05_1.py
19  price |   .0825 |   .0925
20 $ 0.50 | $  0.54 | $  0.55
21 $ 1.00 | $  1.08 | $  1.09
22 $ 1.50 | $  1.62 | $  1.64
23 $ 2.00 | $  2.17 | $  2.19
24 $ 2.50 | $  2.71 | $  2.73
25 $ 3.00 | $  3.25 | $  3.28
26 $ 3.50 | $  3.79 | $  3.82
27 $ 4.00 | $  4.33 | $  4.37
28 $ 4.50 | $  4.87 | $  4.92
29 $ 5.00 | $  5.41 | $  5.46
30 $ 5.50 | $  5.95 | $  6.01
31 $ 6.00 | $  6.50 | $  6.55
32 $ 6.50 | $  7.04 | $  7.10
33 $ 7.00 | $  7.58 | $  7.65
34 $ 7.50 | $  8.12 | $  8.19
35 $ 8.00 | $  8.66 | $  8.74
36 $ 8.50 | $  9.20 | $  9.29
37 $ 9.00 | $  9.74 | $  9.83
38 $ 9.50 | $ 10.28 | $ 10.38
39 $10.00 | $ 10.82 | $ 10.93
40 $ """
```

lab05_2.py

```
 1 #!/usr/bin/env python
 2 """lab05_2.py
 3 2.  Write a Breakfast function that takes five arguments: meat, eggs,
 4 potatoes, toast, and beverage.  The default meat is bacon, eggs are
 5 over easy, potatoes is hash browns, toast is white, and beverage is
 6 coffee.  The function just says:
 7
 8 Here is your bacon and scrambled eggs with home fries and rye toast.
 9 Can I bring you more milk?
10
11 Call it at least 3 different times, scrambling the arguments.
12 """
13
14 def Breakfast(meat="bacon", eggs="over easy",
15               potatoes="hash browns", toast="white",
16               beverage="coffee"):
17     print "Here is your %s and %s eggs with %s and %s toast." \
18           % (meat, eggs, potatoes, toast)
19     print "Can I bring you more %s?" % beverage
20
21 Breakfast()
22 Breakfast("ham", "basted", "cottage cheese",
23           "cinnamon", "orange juice")
24 Breakfast("sausage", toast="wheat", beverage="chai")
25
26 """
27 $ lab05_2.py
28 Here is your bacon and over easy eggs with hash browns and white toast.
29 Can I bring you more coffee?
30 Here is your ham and basted eggs with cottage cheese and cinnamon toast.
31 Can I bring you more orange juice?
32 Here is your sausage and over easy eggs with hash browns and wheat toast.
33 Can I bring you more chai?
34 $
35 """
```

Notes on Sequence types:

```
str                     tuple                   list
```

These are not keywords but they are builtin type names so don't use them to name your identifiers.

They have a lot in common.

```
a_string = "abc"        a_tuple = (1, 2, 3)     a_list = [1, 2, 3]
```

Accessing **[i]** accesses a particular element of any sequence:

```
a_string[0]             a_tuple[0]              a_list[0]
'a'                     1                       1
```

`a_string[-1]` is the last element: "`c`"

`a_string[-4]` gets an error if it doesn't exist!

Slicing **[:]** makes a new sequence object:

```
>>> twist = "Somebody danced."
>>> twist[4:8]
'body'
>>>
```

- Syntax: **[start_index : almost_end_index]**
- `[:3]` gets first three: `0, 1, 2`
- `[3:]` gets from `[3]` to the end: `3, 4, ..., last_index`
- `[:]` gets from start to end: `0, 1, ..., last_index`

```
>>> buy_it = "wholesale"
>>> print buy_it[:5], buy_it[5:]
whole sale
```

Slicing never gets an error, just an empty object!

Concatonation: **+** is supported for all sequence types:

- Both operands must be the same sequence type.
- The result is always a new sequence of the same type.

Plus augmented assignment **+=** is supported. The rules are:

- `a_string += another_string`
- `a_tuple += another_tuple`
- `a_list += any_sequence_type`

```
>>> some_list = [1, 2, 3]
>>> some_list += [4]
>>> some_list
[1, 2, 3, 4]
>>> some_list += (5, 6)
>>> some_list
[1, 2, 3, 4, 5, 6]
>>> some_list += "abc"
>>> some_list
[1, 2, 3, 4, 5, 6, 'a', 'b', 'c']
```


`+=` is often used to accumulate objects:

```
>>> new_tuple = ()
>>> for i in range(5):
...     new_tuple += i,
...
>>> print new_tuple
(0, 1, 2, 3, 4)
```

For sort of this task, you must know how to create an empty sequence of each type. (Next page.)

Empty sequence:

```
''                    ()                    []
```

Sometimes you want to create a one-element, or singleton sequence, to start. Here are singletons of each type:

Singleton sequence:

```
'1'                   1,                    [1]
```

The one-element tuple is the interesting one:

```
>>> b_tuple
('d', 'e')
>>> b_tuple += 'f'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can only concatenate tuple (not "str") to tuple
>>> b_tuple += 'f',
>>> b_tuple
('d', 'e', 'f')
>>>
```


***Repetition* *** is supported for all three sequence types:

```
>>> twister = ('wood', 'chuck')
>>> 3 * twister
('wood', 'chuck', 'wood', 'chuck', 'wood', 'chuck')

>>> call = "kitty "
>>> 3 * call
'kitty kitty kitty '

>>> octave = ['do', 're', 'mi', 'fa', 'so', 'la', 'ti']
>>> 2 * octave
['do', 're', 'mi', 'fa', 'so', 'la', 'ti', 'do', 're', 'mi', 'fa', 'so', 'la', 'ti']
```

in is good for testing membership. The result is `True` or `False`:

```
>>> neighbors = ['Amy', 'Joe']
>>> 'Amy' in neighbors
True

>>> if 'Alice' not in neighbors:
...     neighbors += ['Alice']
...
>>> neighbors
['Amy', 'Joe', 'Alice']
```

in is also coupled with ***for*** to iterate any sequence:

```
>>> for each in neighbors:
...     print each,
...
Amy Joe Alice
>>>
```

Builtin functions: `len, max, min, sum, cmp, str, repr`:

`len(seq), max(seq), min(seq), sum(seq)` do exactly what you'd expect.

`cmp(seq1, seq2) == 0` if they are exactly the same type and the same values.

`str(seq)` and `repr(seq)` make printable representations of the sequence. `str` provides a friendly string; `repr` provides a string that can be fed into a call to `eval` which returns a new seq that is `==` to the original sequence:

```
>>> repr((1, 2))
'(1, 2)'
>>> eval(repr(c_tuple))
(1, 2)
>>>
```

Factory Functions:

`tuple(seq)` makes a tuple of any sequence:

```
>>> c_string = "cloud"
>>> tuple(c_string)
('c', 'l', 'o', 'u', 'd')
```

`list(seq)` makes a list of any sequence:

```
>>> list(c_string)
['c', 'l', 'o', 'u', 'd']
```

`str(seq)` makes a str of any sequence:

```
>>> str(list(c_string))
"['c', 'l', 'o', 'u', 'd']"
```

What's different?

- A string can only have characters as members while a tuples and lists can have any object as a member.
- Only lists support item assignment:

  `a_list[2] = 'a'  is good.`

  But:

  `a_tuple[2] = 'a' is an error.`

  and

  `a_string[2] = 'a' is an error.`

  Note: This attribute of a list is called *mutability*:

  - **tuples** are not mutable.
  - **strings** are not mutable.
  - **lists** are mutable.
- Each sequence type has a different list of builtin functions available.

Strings have lots of builtin functions:

```
>>> dir('')                          [or dir(a_string) or dir(str)]

['__add__', '__class__', '__contains__', '__delattr__', '__eq__',
'__ge__', '__getattribute__', '__getitem__', '__getslice__', '__gt__',
'__hash__', '__init__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__repr__', '__rmul__',
'__setattr__', '__str__', 'capitalize', 'center', 'count', 'decode',
'encode', 'endswith', 'expandtabs', 'find', 'index', 'isalnum',
'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'replace', 'rfind', 'rindex',
'rjust', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper']
```

Magic functions, i.e. those whose names start and end with __ are not meant for us – yet.

Tuples have no functions for us to use:

```
>>> dir(a_tuple)
['__add__', '__class__', '__contains__', '__delattr__', '__eq__',
'__ge__', '__getattribute__', '__getitem__', '__getslice__', '__gt__',
'__hash__', '__init__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__repr__', '__rmul__',
'__setattr__', '__str__']
```

Lists have a few functions in common with strings, and a few of their own:

```
>>> dir(a_list)

['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__eq__', '__ge__', '__getattribute__', '__getitem__',
'__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__repr__', '__rmul__', '__setattr__',
'__setitem__', '__setslice__', '__str__', 'append', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Lists and Strings have `count` and `index` in common:

```
>>> help(a_list.count)
Help on built-in function count:

count(...)
    L.count(value) -> integer -- return number of occurrences of value

>>> help(a_string.count)
Help on built-in function count:

count(...)
    S.count(sub[, start[, end]]) -> int

    Return the number of occurrences of substring sub in string
    S[start:end].  Optional arguments start and end are
    interpreted as in slice notation.


>>> help(a_list.index)
Help on built-in function index:

index(...)
    L.index(value, [start, [stop]]) -> integer -- return first index of value

>>> help(a_string.index)
Help on built-in function index:

index(...)
    S.index(sub [, start [, end]]) -> int

    Like S.find() but raise ValueError when the substring is not found.
```

Only lists have `insert()`. It is important.

```
>>> help(a_list.insert)
Help on built-in function insert:

insert(...)
    L.insert(index, object) -- insert object before index

>>> a_list
[1, 2]
>>> a_list.insert(1, 3)
>>> a_list
[1, 3, 2]
>>>
```

Only lists have `pop, remove, reverse`, and `sort`. This makes sense because these functions alter the list, and only lists can be altered.

```
>>> help(a_list.pop)
Help on built-in function pop:

pop(...)
    L.pop([index]) -> item -- remove and return item at index (default last)

>>> help(a_list.remove)
Help on built-in function remove:

remove(...)
    L.remove(value) -- remove first occurrence of value


>>> help(a_list.reverse)
Help on built-in function reverse:

reverse(...)
    L.reverse() -- reverse *IN PLACE*

>>> help(a_list.sort)
Help on built-in function sort:

sort(...)
    L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
    cmp(x, y) -> -1, 0, 1
```

The "`*IN PLACE*`" on the `list.sort` and `list.reverse` functions means that you must do this:

```
>>> some_list = [2, 1, 0, 5, 3, 6, 8, 7, 9, 4]
>>> some_list.sort()
>>> print some_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

If, instead, your write a line like this:

```
>>> some_list = some_list.sort()
>>> print some_list
None
>>>
```

you threw away your list!!!

Happily, there is another builtin function that works on all sequence types, `sorted(sequence)`. It works on all sequence types because it makes a copy of your sequence, which can be expensive (space-wise). But, you can do this:


```
>>> ride = "cab"
>>> print sorted(ride)
['a', 'b', 'c']
>>>
```

Both `L.sort` and `sorted(sequence)` have an optional argument, `key`:

```
>>> help(sorted)
Help on built-in function sorted in module __builtin__:

sorted(...)
    sorted(iterable, cmp=None, key=None, reverse=False) --> new sorted list
```

The optional `key` argument is a *call-back* function, meaning that `sort` will call it, and sort by the return value. The sort function, however, will return the original list, sorted by the keys.

Providing a homemade `key` function can be very handy. Here comes an example of how to do it.

key_sort.py

```
 1 #!/usr/bin/env python
 2 """ key_sort.py  Demonstrating a homemade key function
 3 for sorting a list."""
 4
 5 def EvensFirstKey(number):
 6     """A key function that will sort all even numbers before
 7     all odd numbers, because "even" sorts before "odd".
 8     """
 9     if number % 2:  # odd
10         return ('odd', number)
11     return ('even', number)
12
13 some_list = [2, 1, 0, 5, 3, 6, 8, 7, 9, 4]
14 print some_list
15 print "After regular sort", sorted(some_list)
16 print "After EvensFirstKey", sorted(some_list, key=EvensFirstKey)
17
18 """
19 $ key_sort.py
20 [2, 1, 0, 5, 3, 6, 8, 7, 9, 4]
21 After regular sort [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
22 After EvensFirstKey [0, 2, 4, 6, 8, 1, 3, 5, 7, 9]
23 $ """
```

Immutable? Look at this:

```
>>> b_list = [2]
>>> b_tuple = 1, b_list, 3
>>> b_list[0] = 22
>>> b_tuple
(1, [22], 3)
>>> b_tuple[1][0]=2
>>> b_tuple
(1, [2], 3)
>>>
```

Note that:

```
c_list = b_list
```

creates a reference to the old object so changing an element in one changes the element in the other.

So if you need an independent copy, use slicing:

```
c_list = b_list[:]
```


Stuff like this is fine:

```
>>> c_tuple = 1, 2, 3
>>> x, y, z = c_tuple
>>> print x, y, z
1, 2, 3
>>> a, b, c = "abc"
>>> print a, b, c
a b c
>>> x, y = y, x
>>> print x, y
2, 1
```

Lab 06

1. Predict the output:

```
>>> says = "Hello Moon"
>>> print says[6:]

---------------

>>> print says[:6] + "World"

---------------

>>> print says[-4:]

---------------

>>> print says[:]

---------------
```


   Now give it a try in the interpreter.

2. (Adapted from Deitel 5.3) Use a list to solve the following problem: Ask the user for 5 numbers, one at a time. As each number is read, print it only if it is not a duplicate of a number already read.

3. You have a list of strings representing names:

```
["Jack Sparrow", "George Washington", "Tiny Sparrow",
 "Jean Ann Kennedy"]
```

   Sort them by last name and print them, last name first. The output should be:

```
Kennedy, Jean Ann
Sparrow, Jack
Sparrow, Tiny
Washington, George
```

   Hints: A function that takes in the name and returns a string with the last name first is doubly useful here. `str.split()` and `str.join()` help with string manipulation. In fact, these two string methods are often useful. Ask more, they are not so intuitive.

4. (Optional) Write a program that inputs a line of text and translates it to *Pig Latin*. The rules for forming Pig Latin word are:

   - If the word begins with a vowel, add "way" to the end of the word.
   - If the word begins with a consonant, extract the consonants up to the first vowel, move those consonants to the end of the word, and add "ay".

   If your prompt is "Tell me something", the output looks like:

   ```
   Tell me something: Tell all
   ellTay allway
   ```

   Have time? Fix it up so that a period, comma, colon, or semicolon at the end of a word doesn't mess you up:

   ```
   $ lab06_4.py

   Tell me something: La Negrita has eyes of flying paper.
   aLay egritaNay ashay eyesway ofway yingflay aperpay.
   $
   ```

   More time? Those upper-case letters are misplaced. Fix that!

   Note: My solution will not be presented until Lab 8.