lists-and-loops

February 8, 2015

Contents

0.1 Lists and loops

In this session we will learn how to store data in lists and how to use for loops plus some basic flow control

So what are lists anyway? Lists are a compound data type use to group together other values. Let's give it a try:

```
In [1]: list1 = ["first", "second", "third"]
```

Here we created a new list using the square brackets `[]`, we could also use the `list()` function

```
In [3]: list2 = [1, 2, 3]
```

```
In [4]: list3 = ["pi", 3.1415926, "answer", 42]
```

```
In [5]: emtpy_list = []
```

We created four lists with different content. Some important things to know about lists are:

- Values don't have to be of the same type (see `list3`)
- Lists can be **modified** after creation (they are *mutable*)
- Elements can be changed, added and deleted
- Lists are versatile and are used extensively in typical Python code

0.1.1 Accessing elements in a list

Lists are **indexed** and elements in a list can be accessed via their index!

```
In [9]: len(list1) # How long is the list?
```

```
Out[9]: 3
```

```
In [8]: list1
```

```
Out[8]: ['first', 'second', 'third']
```

The list contains three items, thus the length of the list is three. However, the index of a list starts at **zero**!

```
In [11]: list1[0]
```

```
Out[11]: 'first'
```

```
In [12]: list1[1]
```

```
Out[12]: 'second'
```

```
In [14]: list1[2]
```

```
Out[14]: 'third'
```

A very nice feature of Pythin is that negative indexes go back from the end:

```
In [15]: list1[-1]
```

```
Out[15]: 'third'
```

Trying to access an element in a list that is out of range throws an `IndexError`

```
In [17]: list1[len(list1)]
```

```
    ---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

    <ipython-input-17-413c63c52f02> in <module>()
----> 1 list1[len(list1)]


    IndexError: list index out of range
```

0.1.2 Changing Items

Items can be modified by using the [] notation on the left-hand side of an assignment:

```
In [18]:  print("{0} is roughly {1}".format(list3[0], list3[1])
```

```
pi is roughly 3.141593
```

```
In [19]: list3[1] = 3.14
```

```
In [20]: print("{0} is roughly {1}".format(list3[0], list3[1])
```

```
pi is roughly 3.140000
```

Assigning to a position past the end of the list also throws an IndexError

```
In [21]: list1[3] = "fourth"
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

<ipython-input-21-b7ec03d21090> in <module>()
----> 1 list1[3] = "fourth"


IndexError: list assignment index out of range
```

0.1.3 Appending and Extending

Remember that lists are mutable? This means that we can add and remove items from a list!

```
In [22]: len(list1)
```

```
Out[22]: 3
```

```
In [23]: list1.append("fourth")
```

```
In [24]: list1[-1] # An index of -1 acceses the last element of a list!
```

```
Out[24]: 'fourth'
```

```
In [25]: len(list1)
```

```
Out[25]: 4
```

A list of items can be appended to the end of a list using the `extend` function.

```
In [26]: list1.extend(["fifth", "sixth"])
```

```
In [27]: list1
```

```
Out[27]: ['first', 'second', 'third', 'fourth', 'fifth', 'sixth']
```

0.1.4 Removing Items

Items at arbitrary positions can be removed using the `del` statement:

```
In [28]: list1[3]
```

```
Out[28]: 'fourth'
```

```
In [29]: del list1[3]
```

```
In [30]: list1[3]
```

```
Out[30]: 'fifth'
```

```
In [31]: len(list1)
```

```
Out[31]: 5
```

Alternativel, one can use the `pop()` member function:

```
In [34]: list1.pop(0)

Out[34]: 'first'

In [35]: list1

Out[35]: ['second', 'third', 'fifth', 'sixth']
```

0.1.5 Inserting New Items

The insert member function can be used to insert new items at arbitrary positions in the list

```
In [75]: list1.insert(0, 'first')
         list1.insert(3, 'fourth')
         list1

Out[75]: ['first', 'a', 'b', 'fourth', 'c']
```

Performance note: appending items to the end of the list using append is more efficient (O(1) vs O(N))
Technical detail: Python lists are implemented as dynamically re-sizing arrays (not linked lists as one might expect)

0.1.6 Concatenating lists

Lists can be concatenated using the `+` operator

```
In [74]: list4 = list2 + list3
         list4

Out[74]: [1, 2, 3, 'pi', 3.14, 'answer', 42]
```

0.1.7 Slicing lists

- A copy of a sublist can be created using the `slice` operator `list[start:end]`
- The element at the index `start` is included, the one at `end` is excluded
- Both indices can be ommitted, defaulting to the start and end of the list respectively

```
In [42]: list1[0:1]

Out[42]: ['first']

In [43]: list1[1:3]

Out[43]: ['second', 'third']

In [44]: list1[:2]

Out[44]: ['first', 'second']

In [45]: list1[1:]

Out[45]: ['second', 'third', 'fourth', 'fifth', 'sixth']

In [46]: list1[:]
```

```
Out[46]: ['first', 'second', 'third', 'fourth', 'fifth', 'sixth']
```

Slice notation can also be used on the left-hand side of an assignment to replace multiple elements

```
In [47]: list1[0:2] = ['#1', '#2']

In [48]: list1

Out[48]: ['#1', '#2', 'third', 'fourth', 'fifth', 'sixth']

In [49]: list1[4:6] = []

In [50]: list1

Out[50]: ['#1', '#2', 'third', 'fourth']
```

0.1.8 Variables and References

- In Python, all variables are references
- The = operator does not copy an object, it just creates another reference (name) for the **same object**!

```
In [51]: list1

Out[51]: ['#1', '#2', 'third', 'fourth']

In [52]: list5 = list1

In [53]: list5

Out[53]: ['#1', '#2', 'third', 'fourth']

In [54]: list5.pop()

Out[54]: 'fourth'

In [55]: list5

Out[55]: ['#1', '#2', 'third']

In [56]: list1

Out[56]: ['#1', '#2', 'third']

In [58]: print(id(list5))
         print(id(list1))

4404467400
4404467400
```

Both lists ***refer*** to the same list!

0.1.9 Copying

Copy using the slice notation (bad style):

```
In [60]: list5 = list1[:]

In [62]: list5

Out[62]: ['#1', '#2', 'third']

In [63]: list5.pop()

Out[63]: 'third'

In [64]: list5

Out[64]: ['#1', '#2']

In [65]: list1

Out[65]: ['#1', '#2', 'third']

In [61]: print(id(list5))
         print(id(list1))

4404577992
4404467400
```

We **copyed** the list, list5 and list1 ***refer*** to different lists!

0.1.10 The `range()` function

- One very commonly needed list is the list of consecutive integers (e.g. for looping – more about that in a bit)
- The built-in `range([start], end)` function creates such lists

```
In [78]: my_range = range(10)
         my_range

Out[78]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [68]: range(2,5)

Out[68]: [2, 3, 4]
```

0.2 `For` Loops

- Many programming tasks require repeating similar operations several times
- Example: Perform some computation for each data point in some data set
- This can be accomplished using a so-called **for loop**
- In Python, for loops allow repeating operations for each item in a list

```
In [77]: letters = ['a', 'b', 'c']
         letters

Out[77]: ['a', 'b', 'c']
```

In order to loop over the letters we do this:

```
In [81]: for l in letters:
             print(l)
```

```
a
b
c
```

We can also use the range function to loop:

```
In [82]: for i in range(5):
             print(i)
```

```
0
1
2
3
4
```

Antother way to loop over a list is to use the range function to get a range of indexes for a list and then access each element by index:

```
In [84]: for i in range(len(letters)):
             print(letters[i]) # Access element at that index
```

```
a
b
c
```

Because looping over a `list` created by `range()` is so common, Python provides an optimized version called `xrange()` Use this whenever you would use `range()` in a for loop (it is much faster!)

```
In [89]: for x in xrange(5):
             print(x)
```

```
0
1
2
3
4
```

Technical detail: `xrange()` does not create a `list` but returns a so-called ***iterator***, which can be looped over using `for`

For loops can be used to do repeated calculations:

```
In [16]: for celsius in range(-5, 5):
             fahrenheit = celsius * 9.0/5.0 + 32
             print("{0} C {1:.2f} F".format(celsius, fahrenheit))
```

```
-5 C 23.00 F
-4 C 24.80 F
-3 C 26.60 F
-2 C 28.40 F
-1 C 30.20 F
0 C 32.00 F
1 C 33.80 F
2 C 35.60 F
3 C 37.40 F
4 C 39.20 F
```

0.3 Flow control: If Statements

- Another fundamental concept in imperative programming are conditionals
- Using conditionals one can write code that performs different computations depending on whether a condition is `True` or `False`
- In Python (and many other programming languages), such statements are formed using the if keyword as follows:

```
In [92]: if True:
             print("Yes, it is true!")
```

```
Yes, it is true!
```

```
In [94]: if 1 == 0:
             print("Something is very wrong here!")
```

```
In [95]: if len(letters) < 5:
             print("Oh my, such a short list!")
```

```
Oh my, such a short list!
```

`if` statements are particularly useful when they are combined with loops

```
In [96]: for x in xrange(10):
             if x % 2 == 0:
                 print(x, "is even!")
```

```
0 is even!
2 is even!
4 is even!
6 is even!
8 is even!
```

There is also an extended form of the `if` statement that contains an `else` block, which is executed whenever the condition is `False`

```
In [99]: for x in xrange(10):
             if x % 2 == 0:
                 print(x, "is even!")
             else:
                 print(x, "is odd!")
```

```
0 is even!
1 is odd!
2 is even!
3 is odd!
4 is even!
5 is odd!
6 is even!
7 is odd!
8 is even!
9 is odd!
```

```
In [101]: for x in xrange(6):
              if x % 2 == 0:
                  print(x, "is even!")
              elif x == 3:
```

```
            print("Three is a great number!")
        elif x == 5:
            print("High five!")
        else:
            print(x, "is odd!")
```

```
0 is even!
1 is odd!
2 is even!
Three is a great number!
4 is even!
High five!
```

You commonly want to create a new list by transforming and filtering an old list:

```
In [103]: even_numbers = []
          for x in xrange(10):
              if x % 2 == 0:
                  even_numbers.append(x)
          print even_numbers
```

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

Because this is such a common operation Python also provides a shortcut in form of ***list comprehensions*** (they are truely great!)

```
In [106]: even_numbers = [x for x in xrange(10) if x % 2 == 0]
          print even_numbers
```

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

0.3.1 Comparison Operators

Numeric types can be compared using the comparison operators, which yield `True` or `False` The operators are:

- a < b | a less than b
- a > b | a greater than b
- a == b | a equal to b
- a <= b | a less than or equal to b
- a >= b | a greater than or equal to b

```
In [107]: 1 == 1
```

```
Out[107]: True
```

```
In [108]: 1 == 0
```

```
Out[108]: False
```

```
In [109]: 5 > 1
```

```
Out[109]: True
```

```
In [110]: 5 >= 5
```

```
Out[110]: True
```

```
In [111]: True == False

Out[111]: False
```

The boolean values True and False can be combined using the boolean operators and, or, and not

```
In [113]: a = True
          b = False

In [115]: not b

Out[115]: True

In [116]: a and b

Out[116]: False

In [117]: a or b

Out[117]: True

In [118]: a and not b

Out[118]: True

In [119]: (1==0) or (3>2) and a

Out[119]: True
```

Any expression that evaluates to a boolean value can be used as condition for an `if` statement

0.3.2 Types

- As we have seen, objects in Python can have different types
  - numeric types (int, float)
  - strings (str)
  - sequence types (tuple, list)
  - boolean type (bool)
- Another fundamental type (dict) will be covered tomorrow
- In addition to these built-in types, users can define their own types (more on that tomorrow)
- Python is dynamically typed (the type of object a variable refers to can change throughout a program)

```
In [120]: type(42)

Out[120]: int

In [121]: type(3.14)

Out[121]: float

In [122]: type("Hello world!")

Out[122]: str

In [123]: type([])

Out[123]: list
```

```
In [124]: type(True)

Out[124]: bool

In [2]: number = 42
        print(number, "is a ", type(number))
        number = number + 2.5
        print(number, "is a " ,type(number))#

42 is a  <type 'int'>
44.5 is a  <type 'float'>
```