CS61A Lecture 11

Monday, September 23rd, 2019

We have so far been taking in numbers and integers and operating on them, but many computer programs work on more than just numbers. Instead, many programs use data, and we use lists and containers to take care of data.

Lists

A built-in data type in Python is a list, which is a square bracket followed by the values separated by commas.

```
[41, 43, 45, 47]
```

There are functions that can operate on lists, such as `len` :

```
>>> len([41,43,45,47])
4
```

You can also pull individual elements out using their index. As with other things in computing, the first value in the sequence is indexed 0.

```
>>> [41,43,45,47][0]
41
```

You can put whatever you want in a list, including integers, strings, and expressions. You will always end up with a sequence of values, not the expressions themselves.

In the `operator` module, you can also use the `getitem` function.

It is confusing that square brackets both start a list and specify its index. Remember that if you don't already have a list, square brackets start a list; if you do, it tells you which index to look at.

Concatenation and repetition

You can make lists longer by simply adding and multiplying them:

```
>>> [2,7]+2*[1,8,2,8]
[2,7,1,8,2,8,1,8,2,8]
```

You cannot however subtract and divide.

Containers

There is a built-in operator for testing whetehr an element appears in a compound value.

```
>>> 1 in [1,8,2,8]
```

```
True
>>> 3 not in [1,8,2,8]
True
```

You cannot check whether two values are a subset of a larger list:

```
>>> [1,8] in [1,8,2,8]
False
>>> [1,8] in [4,[1,8],2]
True
>>> 1 in [2,[1,8],4]
False
```

For statements

A new kind of control statement that makes programming easier by not having to track indices.

Let's define a new function:

```
def count(s,v):
    """Returning how many times V appears in S.
    >>>count([1,8,2,8],2)
    1
    >>>count([1,8,2,8],2)
    2"""
```

How would we write this with while?

```
    total, index = 0,0
    while index < len(s):
        if s[index] == v:
            total = total + 1
        index = index + 1
    return total
```

And how would we write this with for?

```
    total = 0
    for elem in s:
        if elem == v:
            total += 1
    return total
```

Is the `in` above the same `in` as the operator above? No. It is the same word used in a different context.

> **Do for statements only work with lists?**
> No. They work on all containers, even if lists are the only data type we've seen so far.

For Statement Execution Procedure

1. Evaluate the header `<expression>`, which must yield an iterable value (sequence).
2. For each element in that sequence, in order:
   a. Bind `<name>` to that element in the current frame.
   b. Execute the `<suite>`.

> **The zip function**
> Zip takes two lists and zig-zags them.
>
> ```
> >>> for x in zip([1,3,5],[2,4,6]):
> ...   print(x)
> ...
> 1 2
> 3 4
> 5 6
> ```

The Range Type

A range is a sequence of consecutive integers. Ranges can actually represent more general integer sequences. Range lets you get a sequence of numbers starting at x up until the last number before y.

So thus:

```
>>> list(range(-2,2))
[-2,-1,0,1]
```

Calling range by itself will result in just the range being displayed.

```
>>> range(-2,2)
2
```

Remember that `range` doesn't go until the upper bound! The length of the list is its start point subtracted from its endpoint. The element selection is the starting value added to the index.

You can make your life easier by just specifying the endpoint, and Python will automatically set the start point to 0.

How do we sum numbers up until the last integer before n?

We could:

```
def sum_below(n):
    total = 0
    for i in  range(n):
        total += 1
    return total
```

Python programmers commonly use `_` as the variable name for a variable in a for statement that is never used:

```
>>> for _ in range(3):
```

```
...     print('Go Bears!')
Go Bears!
Go Bears!
Go Bears!
```

List Comprehension

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'n', 'o', 'p']
>>> [letters[i] for i in [3, 4, 6, 8]]
[d, e, m, o]
```

Many sequence processing operations can be expressed by evaluating a fixed expression for each element in a sequence and collecting the resulting values in a result sequence. In Python, a list comprehension is an expression that performs such a computation.

```
[<map exp> for <name> in <iter exp> if <filter exp>]
```

We can shorten it to:

```
[<map exp> for <name> in <iter exp>]
```

Historically, `<map exp>` just tells you how to define new values, called map expressions.

Here is an example of list comprehension:

```
>>>[i+1 for i in odds if 25 % i == 0]
2,6
```

The nice thing about list comprehensions is that they open a new frame so they don't affect existing variables in the current frame the list comprehension was run in.

Evaluation Procedure

A combined expression that evaluates to a list using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty result list that is the value of the expression
3. For each element in the iterable value of `<iter exp>` :
   a. Bind `<name>` to that element in the new frame from step 1
   b. If `<filter exp>` evaluates to a true value, then add the value of `<map exp>` to the result list

Strings

Strings are an abstraction. How we represent text under the hood is not exactly our problem. Strings are great because we can use them to represent anything, from data to language to entire programs.

For example:

```
>>> s = 'curry = lambda f: lambda x: lambda y: f(x,y)'
>>> type(s)
str
>>> exec(s)
>>> curry(pow)(2)(3)
8
```

String Literals Have Three Forms

You can use single quotes or double quotes, which are requivalent.

```
>>>'I am string!'
'I am string!'
>>>"I'm string!"
"I'm string!"
>>>"""The Zen of Python
claims,"""
"The Zen of Python \nclaims,"
```

> The backslash is an escape character that allows Python to display characters that would otherwise be difficult to display, such as a new line.