Lists

Manipulating list objects

D.S. Hwang

Department of Software Science
Dankook University

I cannot teach anybody anything.
I can only make them think.
The beginning of wisdom is a definition of terms.

Socrates

Outline

Overview

Each variable has referred to a single number or string.

- How to work with collections of data and use a Python type named `list`
- How to access files and represent their contents using lists

Lists and Indices

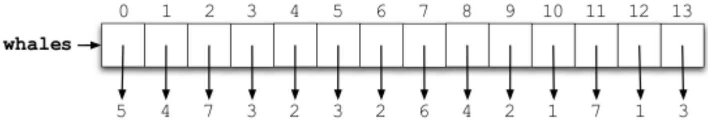
Look at the the table of $\boxed{\text{gray whale}}$ census which were counted near the Coal Oil Point Natural Reserve in a two-week period in the spring of 2008

Day	Number of Whales
1	5
2	4
3	7
4	3
5	2
6	3
7	2
8	6
9	4
10	2
11	1
12	7
13	1
14	3

Lists and Indices

A `list` data structure can be used to store all the values of the table.

```
>>> whales = [5,4,7,3,2,3,2,6,4,2,1,7,1,3]
>>> whales # Number of whales seen per day
[5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
```



Lists and Indices

Using an index, we can get the specific item on the list.

```
>>> whales = [5,4,7,3,2,3,2,6,4,2,1,7,1,3]
>>> whales[0]
5
>>> whales[1]
4
>>> whales[13]
3
```

Use only those indices that are in the range from zero up to one less than the length of the list.

Lists and Indices

Python lets us index backward from the end of a list starting with -1.

```
>>> whales = [5,4,7,3,2,3,2,6,4,2,1,7,1,3]
>>> whales[-1]
3
>>> whales[-2]
1
>>> whales[-14]
5
```

Lists and Indices

An empty list is a list with no items in it.

```
>>> whales = []
>>> whales[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Lists and Indices

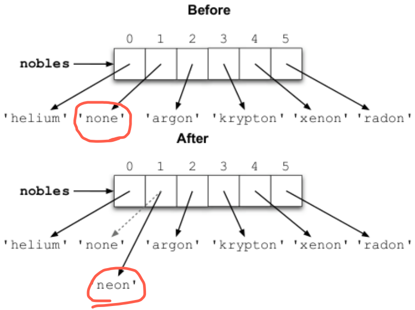
Lists can contain any type of data, including integers, strings, and even other lists.

```
>>> krypton = ['Krypton', 'Kr', -157.2, -153.4]
>>> krypton[-2]
-157.19999999999999
>>> krypton[1]
'Kr'
```

Modifying Lists

What happens when we modify an item on the list.

```
>>> nobles = [’helium’,’none’,’argon’,’krypton’,’xenon’,’radon’ ]
>>> nobles[1]="neon"
>>> nobles
[’helium’, ’neon’, ’argon’, ’krypton’, ’xenon’, ’radon’]
```



Modifying Lists

Lists are mutable.

- Their contents can be changed after they have been created.

Numbers and strings are immutable

- These are not changed after we have created them.
- Methods that appear to, like upper, actually create new strings.

```
>>> name = "dankook"
>>> cap = name.upper()
>>> print(name, cap)
dankook DANKOOK
```

Modifying Lists

`var = L[i]`

- Get the value of the item at location *i* in the list `L`

`L[i] = expr`

- Figure out where item *i* in the list `L` is located so that we can overwrite it.

Built-in Functions on Lists

A few built-in functions in Python

```
>>> half_lives = [87.74, 24110.0, 6537.0, 14.4, 376000.0]
>>> len(half_lives)
5
>>> max(half_lives)
376000.0
>>> min(half_lives)
14.4
>>> sum(half_lives)
406749.14000000001
>>> i = 2
>>> 0 <= i < len(half_lives)
True
```

Built-in Functions on Lists

List concatenation

```
>>> stmt = 'I am a student in Dankook University'
>>> lst = stmt.split()
>>> lst
['I','am','a','student','in','Dankook','University']
>>> lst + "."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
>>> lst + ["."]
['I','am','a','student','in','Dankook','University','.']
```

Processing List Items

Let us process each element in a list using a `for` loop

```
for variable in list:
    block
```

```
>>> velocities = [0.0, 9.81, 19.62, 29.43]
>>> for v in velocities:
...     print ( "Metric:" , v, "m/sec;" "Imperial:" , v * 3.28, "ft/sec" )
...
Metric: 0.0 m/sec; Imperial: 0.0 ft/sec
Metric: 9.81 m/sec; Imperial: 32.1768 ft/sec
Metric: 19.62 m/sec; Imperial: 64.3536 ft/sec
Metric: 29.43 m/sec; Imperial: 96.5304 ft/sec
```

Processing List Items

Nested loop

```
for var_1 in list_1:
    for var_2 in list_2:
        block
```

```
>>> outer = ['Li', 'Na', 'K']
>>> inner = ['F', 'Cl', 'Br']
>>> for metal in outer:
...     for gas in inner:
...         print(metal + gas)
...
LiF
LiCl
LiBr
NaF
NaCl
NaBr
KF
KCl
KBr
```

Processing List Items

```python
# multiplication_table.py
def print_table():
    '''Print the multiplication table for numbers 1 through 5.'''
    numbers = [1, 2, 3, 4, 5]
    # Print the header row.
    for i in numbers:
        print('\t' + str(i))
    print() # End the header row.
    # Print the column number and the contents of the table.
    for i in numbers:
        print(i)
        for j in numbers:
            print('\t' + str(i * j))
        print() # End the current row.
```

Processing List Items

```
>>> from multiplication_table import *
>>> print_table()
    1       2       3       4       5
1       1       2       3       4       5
2       2       4       6       8       10
3       3       6       9       12      15
4       4       8       12      16      20
5       5       10      15      20      25
```

4

Slicing

i,i+1,...,j-1

Access items on lists with sub-indexing

```
list[i:j]
```

```
>>> celegans_markers = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> celegans_markers
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> useful_markers = celegans_markers[0:4]
>>> useful_markers
['Emb', 'Him', 'Unc', 'Lon']
>>> celegans_markers[:4]
['Emb', 'Him', 'Unc', 'Lon']
>>> celegans_markers[4:]
['Dpy', 'Sma']
>>> celegans_copy = celegans_markers[:]
>>> celegans_markers[5] = 'Lvl'
>>> celegans_markers
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Lvl']
>>> celegans_copy
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
```

Slicing

```
>>> celegans_markers[5] = 'Lvl'
>>> celegans_markers
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Lvl']
>>> celegans_copy
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
```

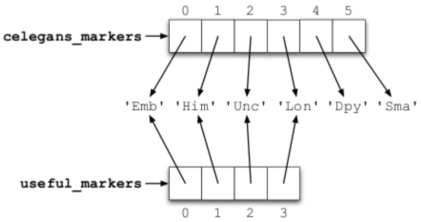


Figure: Slicing doesn't modify lists

Aliasing

Two variables are said to be `aliases` when they refer to the same value

```
>>> celegans_markers = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> celegans_copy = celegans_markers
>>> celegans_markers[5] = 'Lvl'
>>> celegans_markers
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Lvl']
>>> celegans_copy
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Lvl']
```

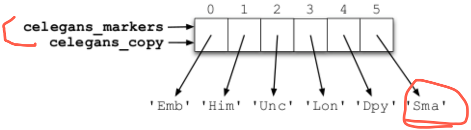


Figure: Aliasing lists

Aliasing

Aliasing is one of the reasons why the notion of mutability is important.

- If `x` and `y` refer to the same list, then any changes to the list through `x` will be "seen" by `y`, and vice versa.
- This can lead to all sorts of hard-to-find errors.
  - A list's value changes, even though your program doesn't appear to assign anything to it.
- This can't happen with immutable values like strings.

Aliasing

Aliasing in function calls

```
>>> def sort_and_reverse(L):
...     L.sort()
...     L.reverse()
...     return L
...
>>> celegans_markers
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Lvl']
>>> sort_and_reverse( celegans_markers )
['Unc', 'Lvl', 'Lon', 'Him', 'Emb', 'Dpy']
```

This function modifies list `L`, and since `L` is an alias of `celegans_markers`, that list is modified as well.

List Methods

Lists are objects and thus have methods.

- These methods in ~~Figure 5.9~~ modify the list instead of creating a new list.
- All of these methods except `pop` return the special value `None`.
  - There is no useful information.
  - There is nothing here.
- Many list methods return `None` rather than creating and returning a new list.

List Methods

- `append` appends a single value, while + expects two lists as operands.
- `append` modifies the list rather than creating a new one.

```
>>> cls.append('purple')
>>> cls
['blue', 'green', 'orange', 'purple', 'red', 'yellow', 'purple']
>>> colors = cls.split()
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'purple']
>>> sorted_colors = colors.sort()
>>> print(sorted_colors)
None
```

Nested Lists

Lists are objects and thus have methods.

- Lists are heterogeneous.
- Lists can contain any type of data.

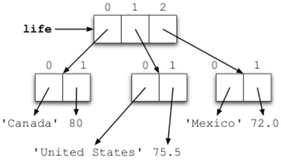


Figure: Nested list

```
>>> life = [['Canada' , 76.5], ['United States' , 75.5], ['Mexico' , 72.0]]
>>> life[2]
['Mexico', 72.0]
>>> life[2][0]
'Mexico'
>>> life[2][1]
72.0
```

Nested Lists

Assigning a sublist to a variable creates an alias for that sublist

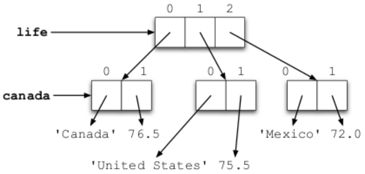


Figure: Aliasing sublist

```
>>> canada = life[0]
>>> canada[1] = 80.0
>>> canada
['Canada', 80.0]
>>> life
[['Canada', 80.0], ['United States', 75.5], ['Mexico', 72.0]]
```

Summary

- Lists are used to keep track of zero or more objects.
- Lists are mutable.
- Slicing is used to create new lists that have the same values or a subset of the values of the originals.
- Two variables refer to the same object by aliasing.
- Tuples are another kind of Python sequence. Tuples are similar to lists, except they are immutable.
- When files are opened and read, their contents are commonly stored in lists of strings.

Problem 1 I

Homework

One of the most important mathematical problems has been to find the area of a polygon. We have a polygon as depicted below.

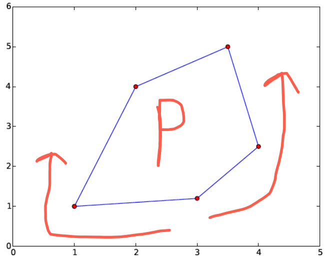


Figure: An example of a polygon

Problem 1 II

Homework

The vertices of polygon P have coordinates $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$, numbered either in a clockwise or counter clockwise way. The area P of the polygon can be computed by just knowing the boundary coordinates:

$$\text{Area of } P = \frac{1}{2}|(x_1y_2 + x_2y_3 + \cdots + x_{n-1}y_n + x_ny_1) - (y_1x_2 + y_2x_3 + \cdots + y_{n-1}x_n + y_nx_1)|$$

Assume that x and y are either lists or arrays. Implement the function to compute the area of a polygon and test your function on a triangular, a quadrilateral, and a pentagon where you can calculate the area by alternative methods for computation.

Problem 2

Homework

Consider two functions $f(x) = x$ abd $g(x) = x^2$ on the interval $[-4, 4]$. Write a program that finds approximately for which values of x the two graphs cross, i.e., $f(x) = g(x)$. Do this by considering N equally distributed points on the interval, at each point checking whether $|f(x) - g(x)| < \epsilon$ with a fixed value ϵ. Let N and ϵ be user input to the program and let the result be printed to screen.

- Run your program with $\boxed{N = 400}$ and $\boxed{\epsilon = 0.01}$.
- Explain the output from your program and try other values of N with fixed ϵ.

Problem 3

Homework

Up through history, great minds have developed different computational schemes for the number π. There are two schemes: one by Leibniz (1646-1716) and one by Euler (1707 - 1783). The scheme by Leibniz may be written

$$\pi = 8 \sum_{k=0}^{\infty} \frac{1}{(4k+1)(4k+3)}$$

while the Euler scheme appears as

$$\pi = \sqrt{6 \sum_{k=1}^{\infty} \frac{1}{k^2}}.$$

If only the first N terms of each sum are used as an approximation to π, each modified scheme will have computed π with some error. Your program should also print out the final error achieved with both schemes, i.e. when the number of terms is N. Run the program with $\boxed{N = 100}$ and explain briefly what the graphs show.

- Write a program that takes N as input from the user.
- Plot the values and errors of both schemes as the number of iterations approaches N when N is 100.
- Explain what your graph shows.