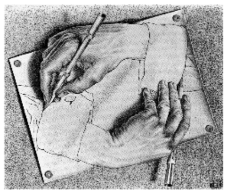
Eric Roberts
CSCI 121

Handout #32
October 10, 2018

Lists

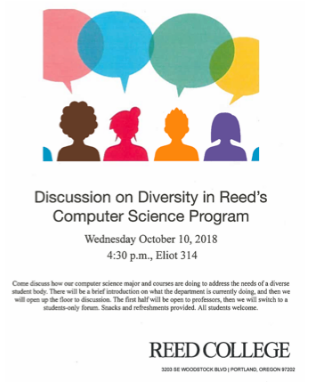
Lists

Eric Roberts
CSCI 121
October 10, 2018



CSCI 121
October 2018

Graphics Contest Winners



Discussion on Diversity in Reed's
Computer Science Program

Wednesday October 10, 2018
4:30 p.m., Eliot 314

Come discuss how our computer science major and courses are doing to address the needs of a diverse student body. There will be a brief introduction on what the department is currently doing, and then we will open up the floor to discussion. The first half will be open to professors, then we will switch to a students-only forum. Snacks and refreshments provided. All students welcome.

REED COLLEGE
3203 SE WOODSTOCK BLVD | PORTLAND, OREGON 97202

Some Missing Bits

- Python supports ***simultaneous assignment,*** which typically has the form

```
x,y,z = a,b,c
```

which simultaneously assigns *a* to *x*, *b* to *y*, and *c* to *z*. All values on the right side of the equal sign are computed before any assignments are made, so that

```
x,y = y,x
```

has the effect of swapping `x` and `y`.

- The Portable Graphics Library exports a `GPoint` type, which encapsulates an *x*-*y* coordinate pair. The type defines the methods `getX` and `getY`.

Arrays and Lists

- From the earliest days of computing, programming languages have supported the idea of an ***array,*** which is an ordered sequence of values.
- The individual values in an array are called ***elements***. The number of elements is called the ***length*** of the array.
- Each element is identified by its position number in the array, which is called its ***index***. In Python—as in almost all modern languages—index numbers begin with 0 and extend up to one less than the length of the array.
- Python implements the array concept in a more general form called a ***list***. Lists support all standard array operations, but also allow insertion and deletion of elements.

Creating Lists in Python

- The simplest way to create a list in JavaScript is to specify its elements surrounded by square brackets and separated by commas. For example, the declaration

```
COIN_VALUES = [ 1, 5, 10, 25, 50, 100 ];
```

creates a constant list of six elements that correspond to the standard coins available in the United States.

- Lists are most commonly represented conceptually as a series of numbered boxes, as in the following representation of `COIN_VALUES`:

`COIN_VALUES`

1	5	10	25	50	100
0	1	2	3	4	5

Nonnumeric Lists

- Lists may contain values of any JavaScript type. For example, the declaration

```
COIN_NAMES = [
   "penny",
   "nickle",
   "dime",
   "quarter",
   "half-dollar",
   "dollar"
];
```

creates the following list:

`COIN_NAMES`

"penny"	"nickel"	"dime"	"quarter"	"half-dollar"	"dollar"
0	1	2	3	4	5

List Selection

- Given an array, you can get the value of any element by writing the index of that element in brackets after the array name. This operation is called ***selection***.
- For example, given the declarations on the preceding slides, the value of `COIN_VALUES[3]` is 25.
- Similarly, the value of `COIN_NAMES[2]` is the string `"dime"`.

`COIN_VALUES`

1	5	10	25	50	100
0	1	2	3	4	5

`COIN_NAMES`

"penny"	"nickel"	"dime"	"quarter"	"half-dollar"	"dollar"
0	1	2	3	4	5

Cycling through List Elements

- One of the most useful idioms for a list is iterating through each of its elements in turn. The standard `for` loop pattern for doing so looks like this:

```
for variable in list:
    Perform some operation on this variable.
```

- As an example, the following function returns the sum of the elements in the list:

```
def sumIntegerList(list):
    sum = 0
    for value in list:
        sum += value
    return sum
```

Sequences

- The last few slides should remind you of the operations for strings, which are almost exactly the same.
- Strings and lists are both examples of a more general class of objects in Python called ***sequences***. All sequences support the following operations:
  - The `len` function
  - Index numbering beginning at 0
  - Negative index numbering that counts backward from the end
  - Selection of an individual element using square brackets
  - Slicing in all its forms
  - Concatenation using the `+` or `+=` operator
  - Repetition using the `*` operator
  - Inclusion testing using the `in` operator

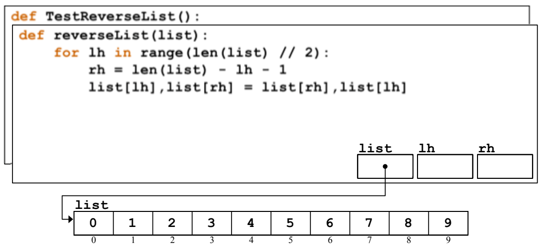
Mutable *vs.* Immutable Types

- The most important difference between a list and a string is that you are allowed to change the contents of a list while the characters in a string are fixed.
- Types like strings for which you are not allowed to change the individual components are defined to be ***immutable***.
- Types like lists where the elements are assignable are said to be ***mutable***.
- Immutable types have many advantages in programming:
  - You don't have to worry about whether values will be changed.
  - Values that are immutable can more easily be shared.
  - Immutable objects are easier to use in concurrent programs.
- Despite these advantages, there are still situations in which mutable types like lists are just the right tools.

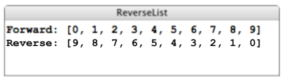
Passing Lists as Parameters

- When you pass a list as a parameter to a function or return a list as a result, only the *reference* to the list is actually passed between the functions.
- The effect of this strategy of representing lists as references is that the elements of a list are effectively shared between a function and its caller. If a function changes an element of a list passed as a parameter, that change will persist after the function returns.
- The next slide simulates a program that does the following:
  1. Initializes a list to contain the integers from 0 to $N-1$.
  2. Prints the elements in the list.
  3. Reverses the elements in the list.
  4. Prints the reversed list on the console.

Trace of the `reverseList` Function



```
def TestReverseList():
def reverseList(list):
    for lh in range(len(list) // 2):
        rh = len(list) - lh - 1
        list[lh],list[rh] = list[rh],list[lh]
```



```
Forward: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Reverse: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Methods that Return Information

list`.index(`*value*`)`	Returns the first index at which *value* appears in *list* or raises an error.
list`.index(`*value*`,` *start*`)`	Returns the first index of *value* after the starting position.
list`.count(`*value*`)`	Returns the number of times *value* appears in *list*.
list`.copy()`	Creates a new list whose elements are the same as the original.

Methods that Add and Remove Elements

list`.append(`*value*`)`	Adds *value* to the end of the list.
list`.insert(`*index*`,` *value*`)`	Inserts *value* before the specified index position.
list`.remove(`*value*`)`	Removes the first instance of *value*, or raises an error if it's not there.
list`.pop()`	Removes and returns the last element of the list.
list`.pop(`*index*`)`	Removes and returns the element at the specified index.
list`.clear()`	Removes all elements from the list.

Methods that Reorder Elements

list`.reverse()`	Reverses the order of elements in the list.
list`.sort()`	Sorts the elements of *list* in increasing order.
list`.sort(`*key*`)`	Sorts the elements of *list* using *key* to generate the key value.
list`.sort(`*key*`,` *reverse*`)`	Sorts in descending order if *reverse* is `True`.

List Methods that Involve Strings

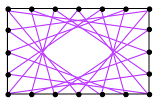
str`.split()`	Splits a string into a list of its components using whitespace as separator.
str`.split(`*sep*`)`	Splits a string into a list using the specified separator.
str`.splitlines()`	Splits a string into separate lines at instances of the newline character.
sep`.join(`*list*`)`	Joins the elements of *list* into a string, using *sep* as the separator.

Exercise: Removing Zero Elements

- Write a function `removeZeroElements(list)` that changes the value of list by removing all elements that are equal to the integer 0.

Lists and Graphics

- Lists turn up frequently in graphical programming. Any time that you have repeated collections of objects, a list provides a convenient structure for storing them.
- As an illustration of the use of lists—and as an example of a pictures using nothing but straight lines—the **`YarnPattern`** program on the next slide simulates the following process:
  - Place pegs at regular intervals around a rectangular border.
  - Tie a piece of yarn around the peg in the upper left corner.
  - Loop that yarn around the peg a certain distance **`DELTA`** ahead.
  - Continue moving forward **`DELTA`** pegs until you close the loop.



Code for the **`YarnPattern`** Program

```
# File: YarnPattern.py

from pgl import GWindow, GLine, GPoint

PEG_SEP = 12                    # The separation between pegs in pixels
N_ACROSS = 80                   # Number of PEG_SEP units horizontally
N_DOWN = 50                     # Number of PEG_SEP units vertically
DELTA = 113                     # Number of pegs to skip on each cycle

GWINDOW_WIDTH = N_ACROSS * PEG_SEP
GWINDOW_HEIGHT = N_DOWN * PEG_SEP

def YarnPattern():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    pegs = createPegs()
    thisPeg = 0
    nextPeg = -1
    while thisPeg != 0 or nextPeg == -1:
        nextPeg = (thisPeg + DELTA) % len(pegs)
        p0 = pegs[thisPeg]
        p1 = pegs[nextPeg]
        line = GLine(p0.getX(), p0.getY(), p1.getX(), p1.getY())
        line.setColor("Magenta")
        gw.add(line)
        thisPeg = nextPeg
```

Code for the **`YarnPattern`** Program

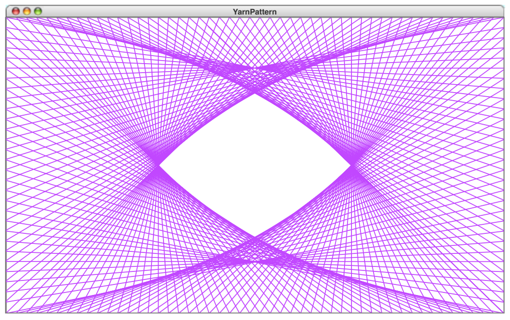
```
# Creates an array of pegs around the perimeter of the graphics window

def createPegs():
    pegs = [ ]
    for i in range(N_ACROSS):
        pegs.append(GPoint(i * PEG_SEP, 0))
    for i in range(N_DOWN):
        pegs.append(GPoint(N_ACROSS * PEG_SEP, i * PEG_SEP))
    for i in range(N_ACROSS, 0, -1):
        pegs.append(GPoint(i * PEG_SEP, N_DOWN * PEG_SEP))
    for i in range(N_DOWN, 0, -1):
        pegs.append(GPoint(0, i * PEG_SEP))
    return pegs

# Startup code

if __name__ == "__main__":
    YarnPattern()
```

A Larger Sample Run


YarnPattern