MORE ABOUT LISTS

LECTURE 04–2

JIM FIX, REED COLLEGE CSCI 121

MONDAY

- We'll start lecture with ***a short quiz***
  - ➡ It will be something like Homework 2:
    - ✦ still some Python scripting (`input` and `print`)
    - ✦ `if` and `if-else` statements
    - ✦ `while` loops

PYTHON LIST SUMMARY

- List creation:

  `[3,1,7]` `[]`
- Accessing its contents:

  `xs[3]` `xs[-1]` `len(xs)`
- Modyifying its contents:

  `xs[3] = 5`
- Changing the length of a list object:

  `xs.append(5)` `xs.extend([8,9,10])` `xs.insert(2,357)`
  `xs.pop()` `del xs[6]`
- Checking membership, content equality, object identity:

  `3 in xs` `xs == [1,2,3]` `xs is ys`
- Scanning using a **`while`** loop:

```
i = 0
while i < len(xs):
    print(xs[i])
    i = i + 1
```

SQUARES FUNCTION

- Given a list of integers, build a new list of its squares.

```
def squares(xs):
    i = 0
    sqxs = []
    while i < len(xs):
        sq = xs[i] ** 2
        sqxs.append(sq)
        i += 1
    return sqxs
```

USING A MORE GENERIC FUNCTION

- Given a list of integers, build a new list with their squares.

```
def all_with_func(xs, f):
    i = 0
    ys = []
    while i < len(xs):
        y = f(xs[i])
        ys.append(y)
        i += 1
    return ys

def squares(ls):

    def square(x):
        return x * x

    return all_with_func(ls,square)
```

SQUARING PROCEDURE

- Given a list of integers, update entries with their squares.

```
def square_all(xs):
    i = 0
    while i < len(xs):
        xs[i] = xs[i] ** 2
        i += 1
```

SQUARING PROCEDURE

- Given a list of integers, update entries with their squares.

```
def square_all(xs):
    i = 0
    while i < len(xs):
        x = xs.pop(i)
        xs.insert(i, x*x)
        i += 1
```

SQUARING PROCEDURE

- Given a list of integers, update entries with their squares.

```
def square_all(xs):
    i = 0
    while i < len(xs):
        x = xs.pop(i)       #      a bit too much
        xs.insert(i, x*x)   # <-- here!
        i += 1
```

SQUARING PROCEDURE

- Given a list of integers, update entries with their squares.

```
def square_all(xs):
    i = 0
    while i < len(xs):
        xs[i] = xs[i] ** 2    # <- just update at i
        i += 1
```

GROUP WORK

- Given a list of integers, split it into evens and odds.

```
def evensOdds(xs):
    ????
```

- It should remove the odds from the list it is given, leaving only the evens.
- It should also return a list of the odds.
- Example use:

```
>>> somelist = [1,2,17,111,8]
>>> evensOdds(somelist)
[1, 17, 111]
>>> somelist
[2, 8]
>>>
```

GROUP WORK SOLUTION

- Given a list of integers, this splits it into evens and odds.

```
def evensOdds(xs):
    odds = []
    i = 0
    while i < len(xs):
        if xs[i] % 2 == 1:
            odds.append(xs[i])
            del xs[i]
        else:
            i = i + 1
    return odds
```

PYTHON LIST SUMMARY

- List creation:

  `[3,1,7]` `[]`
- Accessing its contents:

  `xs[3]` `xs[-1]` `len(xs)`
- Modyifying its contents:

  `xs[3] = 5`
- Changing the length of a list object:

  `xs.append(5)` `xs.extend([8,9,10])` `xs.insert(2,357)`
  `xs.pop()` `del xs[6]`
- Checking membership, content equality, object identity:

  `3 in xs` `xs == [1,2,3]` `xs is ys`
- Scanning using a **`while`** loop:

```
i = 0
while i < len(xs):
    print(xs[i])
    i = i + 1
```

LIST "ARITHMETIC"

- We can build new lists from other list's contents using `+` and `*`:

```
>>> [1,2,17] + [111,8]
[1, 2, 17, 111, 8]
>>> [1,2,17] * 4
[1, 2, 17, 1, 2, 17, 1, 2, 17, 1, 2, 17]
>>> [1,2,17] + []
[1, 2, 17]
>>> [] + [1,2,17]
[1, 2, 17]
>>> [1,2,17] * 1
[1, 2, 17]
>>> [1,2,17] * 0
[]
>>> [] * 4
[]
>>> [] + []
[]
```

SQUARES FUNCTION AGAIN

- Given a list of integers, build a new list of its squares.

```
def squares(xs):
    i = 0
    sqxs = []
    while i < len(xs):
        sq = xs[i] ** 2
        sqxs.append(sq)
        i += 1
    return sqxs
```

REDO OF SQUARES FUNCTION USING +

- Given a list of integers, build a new list of its squares.

```
def squares(xs):
    i = 0
    sqxs = []
    while i < len(xs):
        sq = xs[i] ** 2
        sqxs = sqxs + [sq]
        i += 1
    return sqxs
```

REDO OF SQUARES FUNCTION USING +

- Given a list of integers, build a new list of its squares.

```
def squares(xs):
    i = 0
    sqxs = []
    while i < len(xs):
        sq = xs[i] ** 2
        sqxs = sqxs + [sq]
        i += 1
    return sqxs
```

BROKEN SQUARING PROCEDURE

- This ***does not*** update its list parameter to store its squares***!!!***

```
def square_all(xs):
    i = 0
    old_xs = xs
    xs = []
    while i < len(old_xs):
        sq = old_xs[i] ** 2
        xs = xs + [sq]
        i += 1
```

BROKEN SQUARING PROCEDURE

- This ***does not*** update its list parameter to store its squares***!!!***

```
def square_all(xs):
    i = 0
    old_xs = xs
    xs = []
    while i < len(old_xs):
        sq = old_xs[i] ** 2
        xs = xs + [sq]
        i += 1
```

- Instead it creates a new list object and uses `xs` to refer to it instead.
- The code keeps creating new, longer lists, and uses `xs` to refer to them.

PYTHON TUTOR FOR LISTS

LIST "SLICING"

- We can build new lists by copying portions of other lists:

```
>>> xs = [45,1,8,17,100,6]
>>> xs
[45, 1, 8, 17, 100, 6]
>>> xs[2:5]              # Build a new list from the 2,3,4 slice.
[8, 17, 100]
>>> xs[2:4]              # Build a new list from the 2,3 slice.
[8, 17]
>>> xs[:4]               # Build a new list from the 0,1,2,3 slice.
[45, 1, 8, 17]
>>> xs[4:]               # Build a new list from the 4,5 slice.
[100, 6]
>>> ys = xs[:]           # Build a new list as a full copy.
>>> xs[1] = 121
>>> xs
[45, 121, 8, 17, 100, 6]
>>> ys
[45, 1, 8, 17, 100, 6]
```

EXAMPLE WITH SLICING

- Here is an example where we "slice out" an item:

```
def list_without(xs, index):
    …
```

- Here it is in use:

```
>>> xs = [1,10,100,1000,10000]
>>> list_without(xs, 3)
[1, 10, 100, 10000]
```

EXAMPLE WITH SLICING

- Here is an example where we "slice out" an item:

```
def list_without(xs, index):
    …
```

- Here it is in use. It should not change the list it is given:

```
>>> xs = [1,10,100,1000,10000]
>>> list_without(xs, 3)
[1, 10, 100, 10000]
>>> xs
[1, 10, 100, 1000, 10000]
```

EXAMPLE WITH SLICING

- Here is an example where we "slice out" an item:

```
def list_without(xs, index):
    return xs[:index] + xs[index+1:]
```

- Here it is in use:

```
>>> xs = [1,10,100,1000,10000]
>>> list_without(xs, 3)
[1, 10, 100, 10000]
>>> xs
[1, 10, 100, 1000, 10000]
```

EXAMPLE WITH SLICING

- Here we obtain a list of slices:

```
def all_consecutive_pairs(xs):
    i = 0
    cs = []
    while i < len(xs) - 1:
        cs.append(xs[i:i+2])
        i += 1
    return cs
```

- Here it is in use:

```
>>> xs = [1,10,100,1000]
>>> all_consecutive_pairs(xs)
[[1, 10], [10, 100], [100, 1000]]
```

EXAMPLE WITH SLICING

- Here we obtain a list of slices:

```
def all_consecutive_pairs(xs):
    i = 0
    cs = []
    while i < len(xs) - 1:
        cs.append(xs[i:i+2])
        i += 1
    return cs
```

- Here it is in use:

```
>>> xs = [1,10,100,1000]
>>> all_consecutive_pairs(xs)
[[1, 10], [10, 100], [100, 1000]]
```

EXAMPLE WITH SLICING

- Here we obtain a list of slices:

```
def all_consecutive_pairs(xs):
    i = 0
    cs = []
    while i < len(xs) - 1:
        cs.append(xs[i:i+2])
        i += 1
    return cs
```

- Here it is in use:

```
>>> xs = [1,10,100,1000]
>>> all_consecutive_pairs(xs)
[[1, 10], [10, 100], [100, 1000]]
```

LISTS OF LISTS

- Lists can be stored within other lists.

```
>>> lls = [[45,19],[8],[17,100,6],[]]
>>> lls[2]
[17, 100, 6]
>>> lls[2][0]
17
>>> lls[2][0] = 7777
>>> lls
[[45, 19], [8], [7777, 100, 6], []]
>>> lls[0].pop()
19
>>> lls[0].extend([0,0,0])
>>> lls
[[45, 19, 0, 0, 0], [8], [7777, 100, 6], []]
>>> lls.append([5,4,3,2])
>>> lls
[[45, 19, 0, 0, 0], [8], [7777, 100, 6], [], [5, 4, 3, 2]]
```

MATRIX REPRESENTATION

- A list of lists is often used to represent a matrix (or a table) of values:

```
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
```

- Here is how we might store this data:

```
>>> table = [[0, 1, 2, 3, 4],[5, 6, 7, 8, 9], [10,11,12,13,14]]
>>>
```

MATRIX REPRESENTATION

- A list of lists is often used to represent a matrix (or a table) of values:

```
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
```

- Here is how we might store this data:

```
>>> table = [[0, 1, 2, 3, 4],[5, 6, 7, 8, 9], [10,11,12,13,14]]
>>> print(????)
```

MATRIX REPRESENTATION

- A list of lists is often used to represent a matrix (or a table) of values:

```
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
```

- Here is how we might store this data:

```
>>> table = [[0, 1, 2, 3, 4],[5, 6, 7, 8, 9], [10,11,12,13,14]]
>>> print(table[2][3])
13
>>>
```

MATRIX REPRESENTATION

- A list of lists is often used to represent a matrix (or a table) of values:

```
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
```

- Here is how we might store this data:

```
>>> table = [[0, 1, 2, 3, 4],[5, 6, 7, 8, 9], [10,11,12,13,14]]
>>> print(table[2][3])
13
>>> table[0][1] = 77
```

MATRIX REPRESENTATION

- A list of lists is often used to represent a matrix (or a table) of values:

```
 0 77  2  3  4
 5  6  7  8  9
10 11 12 13 14
```

- Here is how we might store this data:

```
>>> table = [[0, 1, 2, 3, 4],[5, 6, 7, 8, 9], [10,11,12,13,14]]
>>> print(table[2][3])
13
>>> table[0][1] = 77
```

ALL PREFIXES GROUP WORK

- Let's work to make a list of slices:

```
def all_prefixes(xs):
    …
```

- Here it is in use:

```
>>> xs = [1,10,100, 1000]
>>> all_prefixes(xs)
[[], [1], [1, 10], [1, 10, 100], [1, 10, 100, 1000]]
```

ALL PREFIXES SOLUTION

- Here is a sample solution:

```
def all_prefixes(xs):
    pfxs = []
    i = 0
    while i <= len(xs):
        pfxs.append(xs[:i])
        i += 1
    return pfxs
```

- Here it is in use:

```
>>> xs = [1,10,100, 1000]
>>> all_prefixes(xs)
[[], [1], [1, 10], [1, 10, 100], [1, 10, 100, 1000]]
```

ALL PREFIXES SOLUTION #2

- Here is another version. We write it to using +

```
def all_prefixes(xs):
    pfxs = []
    i = 0
    while i <= len(xs):
        pfxs = pfxs + [xs[:i]]
        i += 1
    return pfxs
```

- Here it is in use:

```
>>> xs = [1,10,100, 1000]
>>> all_prefixes(xs)
[[], [1], [1, 10], [1, 10, 100], [1, 10, 100, 1000]]
```

ALL PREFIXES BROKEN SOLUTION

- This version has a bug:

```
def all_prefixes(xs):
    pfxs = []
    i = 0
    while i <= len(xs):
        pfxs = pfxs + xs[:i]
        i += 1
    return pfxs
```

- Here it is in use:

```
>>> xs = [1,10,100, 1000]
>>> all_prefixes(xs)
????
```

ALL PREFIXES BROKEN SOLUTION

- This version has a bug:

```
def all_prefixes(xs):
    pfxs = []
    i = 0
    while i <= len(xs):
        pfxs = pfxs + xs[:i]
        i += 1
    return pfxs
```

- Here it is in use:

```
>>> xs = [1,10,100, 1000]
>>> all_prefixes(xs)
[1, 1, 10, 1, 10, 100, 1, 10, 100, 1000]
```

PYTHON LIST SUMMARY ENHANCED

- List creation via enumeration, concatenation, repetition, slicing:

```
[3,1,7]  []  [1,2]+[3,4,5]  [1,2]*4  xs[3:5]  xs[3:]  xs[:]
```

- Accessing contents by index; list length:

```
xs[3]  xs[-1]  len(xs)
```

- Updating contents by indexed assignment:

```
xs[3] = 5
```

- Modifying/mutating a list object:

```
xs.append(5)    xs.extend([8,9,10])    xs.insert(2,357)
xs.pop()        del xs[6]
```

- Checking membership, content equality, object identity:

```
3 in xs      xs == [1,2,3]     xs is ys
```

- Scan according to index using a **`while`** loop:

```
i = 0
while i < len(xs):
    print(xs[i])
    i = i + 1
```

TWO PRINTING PROCEDURES

- This procedure outputs the contents of a list.

```
def output_using_while(xs):
    i = 0
    while i < len(xs):
        print(xs[i])
        i = i + 1
```

- This procedure also outputs the contents of a list.

```
def output_using_for(xs):
    for x in xs:
        print(x)
```

ONLY THE EVENS

- This function returns a list of the even values:

```
def only_the_evens(xs):
    es = []
    for x in xs:
        if x % 2 == 0:
            es.append(x)
    return es
```

SQUARES REVISTED

- This function returns the list of squares:

```
def squares(xs):
    sqxs = []
    for x in xs:
        sqxs.append(x*x)
    return sqxs
```

PYTHON LIST SUMMARY ENHANCED WITH FOR

- Creating a new list by enumeration, concatenation, repetition, slicing:

```
[3,1,7]  []  [1,2]+[3,4,5]  [1,2]*4  xs[3:5]  xs[3:]  xs[:]
```

- Accessing contents:

```
xs[3]  xs[-1]  len(xs)
```

- Updating contents:

```
xs[3] = 5
```

- Resizing a list object:

```
xs.append(5)    xs.extend([8,9,10])    xs.insert(2,357)
xs.pop()        del xs[6]
```

- Checking membership, content equality, object identity:

```
3 in xs      xs == [1,2,3]     xs is ys
```

- Scan a list using a **`while`** loop.
- Loop through the contents using a **`for`** loop.

MONDAY

- We'll start lecture with ***a short quiz***
  - ➡ It will ask you to write a function that will likely require need
    - ✦ `if` or `if-else` statements
    - ✦ `while` loops