Computer Science Principles

Unit 3: Algorithms and Programming

LECTURE 6: LIST AND LOOPS (U5)

DR. ERIC CHOU

IEEE SENIOR MEMBER

Topics

1. Variables and Assignments
2. Data Abstraction
3. Mathematical Expressions
4. Strings
5. Boolean Expressions
6. Conditionals
7. Nested Conditionals
8. **Iteration**
9. **Developing Algorithms**
10. **Lists**
11. **Binary Search**
12. Calling Procedures
13. Developing Procedures
14. Parts of a Procedure
15. Libraries
16. Random Values
17. Simulation
18. Algorithmic Efficiency
19. Undecidable Problems

Iteration

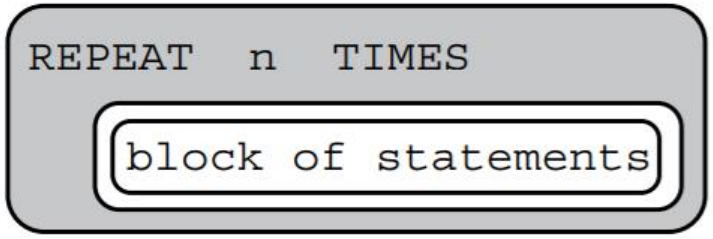
LECTURE 1

Iteration

- **Iterative statements** are also called loops, and they repeat themselves over and over until the condition for stopping is met.
- There are two types of loops.

REPEAT n TIMES loop

- In College Board's Pseudocode, the first is a **REPEAT n TIMES loop**, where the n represents some number.

Text:
`REPEAT n TIMES`
`{`
`<block of statements>`
`}`
Block:
 
	The code in `block of statements` is executed `n` times.

Iteration

- **n** can either be a number outright or some variable. The loop could state outright "REPEAT 5 TIMES," for example, but it's more likely that you'll see something like this:

```
n = 5
REPEAT n TIMES:
   print (n)
```

Python

In Python, the closest thing in comparison is a for... in range loop.

```
for x in range (0, 6):
 print (x)
 #The range includes 0 but not 6, so this loop runs 5 times
 #This loop can also be written as...

for x in range (6):
 print (x)
```

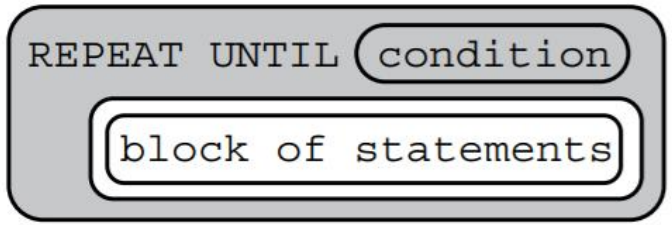
REPEAT UNTIL (condition) loop

- The second type of loop is a **REPEAT UNTIL (condition) loop**, where the loop will continue to run until a condition is met.

Text:

```
REPEAT UNTIL(condition)
{
 <block of statements>
}
```

Block:



The code in `block of statements` is repeated until the Boolean expression `condition` evaluates to `true`.

Python

- In Python, the main form of loop that operates based on a condition is known as a while loop. Unlike a REPEAT UNTIL loop, while loops run "while" a condition is met and end when that condition is no longer true.

```
tacos = 5
while tacos > 0:
    print ("Nom Nom")
    tacos = tacos - 1
#Note: tacos = tacos - 1 can also be written as tacos -= 1.
#The same goes for addition (tacos += 1)
```

How many times will this loop print "Nom Nom?"

- The answer is 5 times!

Iteration Practice Problem

- Can you think of a way to write this loop so that it matches the REPEAT UNTIL format?

```
tacos = 5
while not tacos == 0:
  print ("Nom Nom")
  tacos = tacos - 1
```

- Using the NOT operator, we can set it so that this loop will continue to run until the tacos variable equals 0.
- Notice how, in both of these examples, there's the tacos = tacos - 1 statement after the print statement? This is to reduce the value of the tacos variable. Once tacos is less than 1, the loop stops running.

Iteration Practice Problem

- If we didn't have this statement, the loop would continue repeating forever (or, at least, until your computer ran out of power or resources). These repeating loops are known as infinite loops.

❗If the condition at the beginning of a REPEAT UNTIL loop is already met when you run the loop in your program, the loop won't execute at all.

```
tacos = 0
while tacos > 0:
  print ("Nom Nom")
  tacos = tacos - 1
```

- In the above example, the loop won't run because tacos already equals 0.

More Operators in Loops

- When writing conditions for loops, you can also use the AND and OR operators.
- Here's an example of one...

```
tacos = 5
avocados = 5
while tacos > 0 and avocados > 0:
    print ("Nom Nom")
    tacos = tacos - 1
    avocados = avocados - 1
#technically, you don't need this last line;
#if just tacos = 0, the loop wouldn't run
```

More Operators in Loops

and the other!

```
tacos = 5
quesadillas = 10
while tacos > 0 or quesadillas > 0:
  print ("Nom Nom")
  tacos = tacos - 1
  quesadillas = quesadillas - 1
#this program will print
#"Nom Nom" 10 times. More food for everyone!
```

Developing Algorithms

LECTURE 2

Developing Algorithms

- We've just looked at two different ways to write the same loop in Python. This brings up an important point: different algorithms can be used to achieve the same goals. This is because algorithms are, at their heart, steps to solve a problem, and there are many different ways to solve the same problem.
- Often times, you'll see different algorithms used to solve the same problem, depending on the needs of the programmers and the program.
- With all this variability, it's no wonder that new algorithms are constantly being developed.
- New algorithms can be created from scratch or by combining and modifying algorithms

Examples of Existing Algorithm Types:

- Determining the max. or min. value in a group of two or more numbers
- Solving math problems: calculating sums, averages, etc.
- Determining a robot's path through a maze (route-finding algorithm)
- Compressing data (see Big Idea 2)
- Sorting a list

Lists

LECTURE 3

Basic List Operators

Basic List Operators

Accessing an element by index:

- This operation allows you to single out an element in a list based on its index number. You can then interact with only this element.

```
grocery_list = ["milk", "eggs", "cheese"]

print (grocery_list[0])
```

- The code's output:

```
milk
```

Text:
`aList[i]`
Block:
`aList` `i`
	Accesses the element of `aList` at index `i`. The first element of `aList` is at index `1` and is accessed using the notation `aList[1]`.

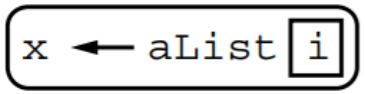
Basic List Operators

Assigning a value of an element of a list to a variable:

- This allows you to assign a variable to a certain element within a list, changing the element. Note that you wouldn't use this operation to add new values to the list, only to change ones already existing.

```
grocery_list = ["milk", "eggs", "cheese"]
change = "soap"
grocery_list[2] = change
print (grocery_list)
```

- The code's output: `["milk", "eggs", "soap"]`

Text: `x ← aList[i]` Block: 	Assigns the value of `aList[i]` to the variable `x`.

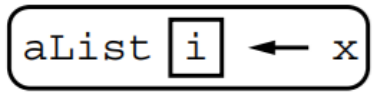
Basic List Operators

Assign a Value to an Element Outright:

- Here's an example in Python:

```
grocery_list = ["milk", "eggs", "cheese"]
grocery_list[2] = "fish"
print (grocery_list)
```

- The code's output: `["milk", "eggs", "fish"]`

Text: `aList[i] ← x` Block: 	Assigns the value of `x` to `aList[i]`.

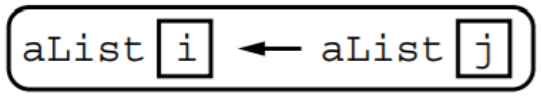
Basic List Operators

Assign the Value of One Element in the List to Another:

- Like this!

```
grocery_list = ["milk", "eggs", "cheese"]
grocery_list[0] = grocery_list[2]
print(grocery_list)
```

- The code's output: `["cheese", "eggs", "cheese"]`

Text: `aList[i] ← aList[j]` Block: 	Assigns the value of `aList[j]` to `aList[i]`.

Adding and Removing Elements

Adding and Removing Elements

Inserting elements at a given index:

Text:
`INSERT(aList, i, value)`
Block:
`INSERT aList, i, value`
	Any values in `aList` at indices greater than or equal to `i` are shifted one position to the right. The length of the list is increased by 1, and `value` is placed at index `i` in `aList`.

Adding and Removing Elements

Inserting elements at a given index:

- This allows you to insert a value into the index position you want. It will increase the length of the list and shift everything greater than or equal to that index down by one place. For example, if you were to insert a new value at the index value 4, what was originally there will move to the index value 5, 5 will move to 6, and so on.

```
grocery_list = ["milk", "eggs", "cheese"]
grocery_list.insert (2, "butter")
print (grocery_list)
```

- The code's output:

```
["milk", "eggs", "butter", "cheese"]
```

Adding and Removing Elements

Adding elements to the end of the list:

Text:
`APPEND(aList, value)`
 	
Block:
`APPEND aList, value`
	The length of `aList` is increased by 1, and `value` is placed at the end of `aList`.

Adding and Removing Elements

Adding elements to the end of the list:

- This allows you to add values to the end of your list.

```
grocery_list = ["milk", "eggs", "cheese"]
grocery_list.append ("flour")
print (grocery_list)
```

- The code's output:

```
["milk", "eggs", "butter", "flour"]
```

Adding and Removing Elements

Removing elements:

Text:
`REMOVE(aList, i)`
Block:
`REMOVE aList, i`
	Removes the item at index `i` in `aList` and shifts to the left any values at indices greater than `i`. The length of `aList` is decreased by 1.

Adding and Removing Elements

Removing elements:

- You can also remove elements.
- In Python, you remove items based on element value rather than index number.

```
grocery_list = ["milk", "eggs", "cheese"]
grocery_list.remove ("eggs")
print (grocery_list)
```

- The code's output: `["milk", "cheese"]`

Adding and Removing Elements

Determining the length of a list:

Text:
`LENGTH(aList)`
Block:
`LENGTH` `aList`
	Evaluates to the number of elements in `aList`.

Adding and Removing Elements

Determining the length of a list:

- This will tell you what the length of your list is.

```
grocery_list = ["milk", "eggs", "cheese"]
print (len (grocery_list))
```

- The code's output: `3`

Looping through Lists

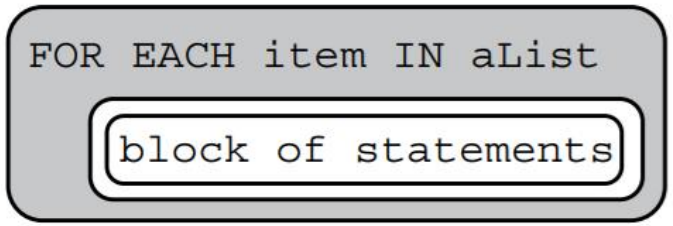
Looping through Lists

- You can also use loops to traverse, or go through, a list. This can either be a complete traversal or a partial traversal, depending on what your loop specifies.

Text:

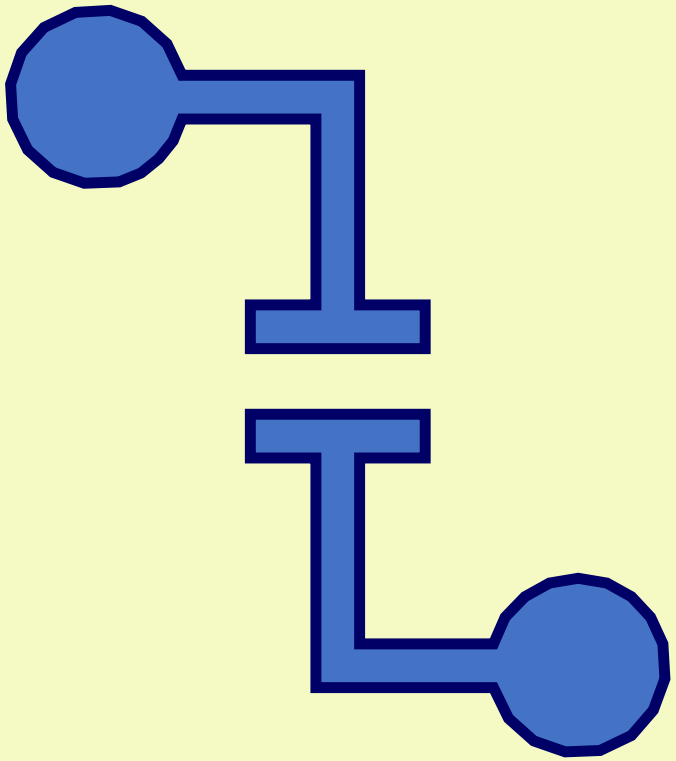
```
FOR EACH item IN aList
{
 <block of statements>
}
```

Block:



The variable `item` is assigned the value of each element of `aList` sequentially, in order, from the first element to the last element. The code in `block of statements` is executed once for each assignment of `item`.

- Common algorithms used with lists will often find the maximum or minimum value inside the list or the average.



Loops

SECTION 1

Loops Explore

LESSON 1 [CODE.ORG]

Loops Investigate

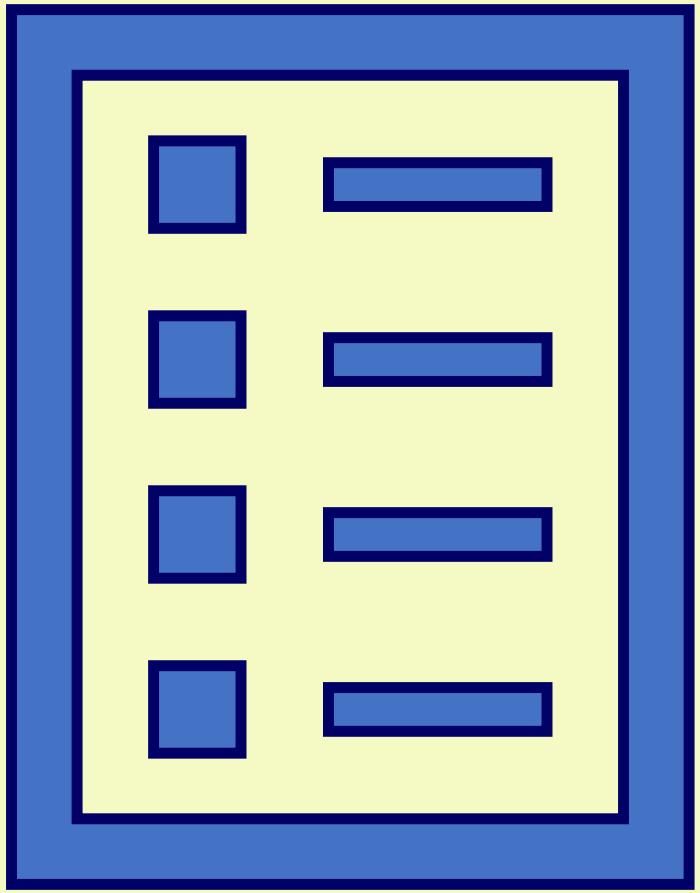
LESSON 2 [CODE.ORG]

Loops Practice

LESSON 3 [CODE.ORG]

Loops Make

LESSON 4 [CODE.ORG]



List

SECTION 2

Lists Explore

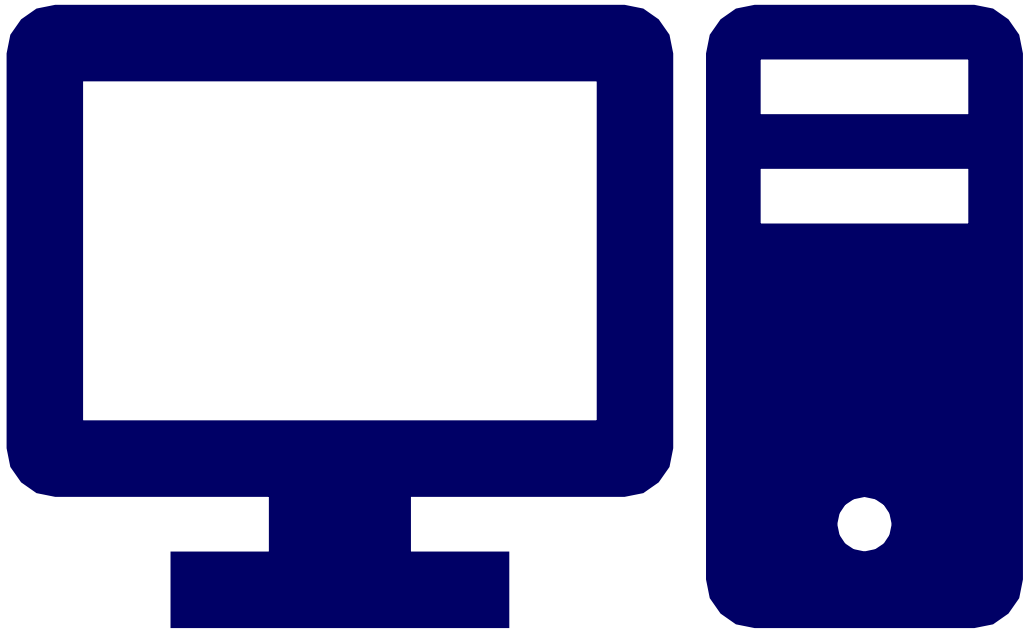
LESSON 5 [CODE.ORG]

Lists Investigate

LESSON 6 [CODE.ORG]

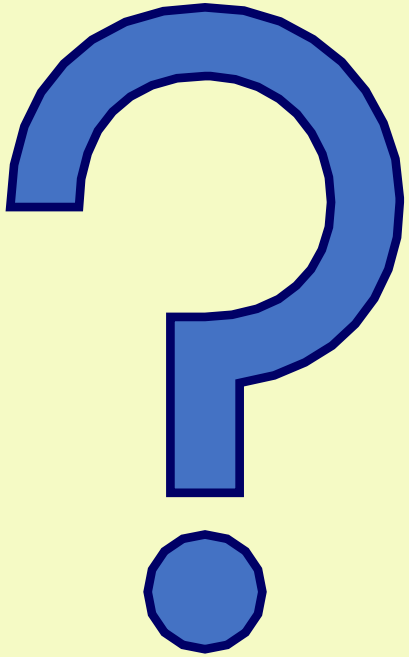
Lists Practice

LESSON 7 [CODE.ORG]



Lists Make

LESSON 8 [CODE.ORG]



Traversal

SECTION 3

Binary Search

LECTURE 4

Binary Search

- The most common way of traversing a list is to go through each item in order, one at a time.
- This is also the most basic way to search through a list. This search method is called a linear or **sequential search** algorithm, and it checks each value of a list in order until the result is found.

Binary Search

- However, this isn't the only way you can search through a list. You can also look through a list using a **binary search**.
- The binary search algorithm starts in the middle of a sorted data set and eliminates half of the data based on what it's looking for. It then repeats the process until the desired value is found or until the algorithm has exhausted all the values in the list.

Traversing a List

- For example, let's say you had a list that looked like this:

```
1, 1, 2, 3, 3, 4, 5, 7, 9, 11, 12
```

and you wanted to find where 12 was.

- If you were doing a binary search, you would divide the list in half and look at the value there, which would be 4.

```
1, 1, 2, 3, 3, 4, 5, 7, 9, 11, 12
```

Traversing a List

- 12 is greater than 4, so the program knows to disregard everything before and including that value.

~~1, 1, 2, 3, 3, 4~~, 5, 7, 9, 11, 12

- The program would then divide the list into half again...

5, 7, 9, 11, 12

Traversing a List

- The program would then divide the list into half again...

  `5, 7, 9, 11, 12`

- and look at the value 9. 9 is less than 12, so the program would eliminate everything before and including that value.

  ~~`5, 7, 9`~~`, 11, 12`

Traversing a List

- This process would go on until the program either found 12 or went through all the values in the list.
- Data must first be *sorted* in order to use a binary search method. However, when used on sorted data, a binary search is often more efficient than a sequential search because it eliminates half the data with each round of splitting. This means that it doesn't have to evaluate many of the results, saving time that the program would usually spend going down the list in a sequential sort. Due to this, the binary search method is commonly used in modern programs.

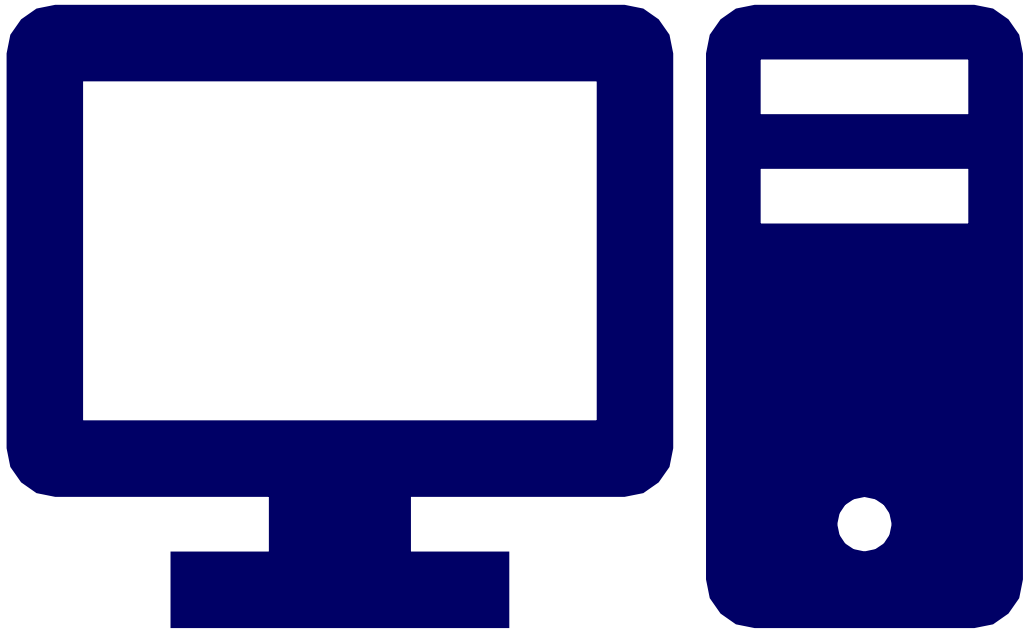
📎 Check out **this video** explaining binary searches, as well as ways they can be written out. To see a binary search algorithm written in Python, go **here**. (Don't worry, you won't need to know how the algorithm works for the test.)

Traversals Explore

LESSON 9 [CODE.ORG]

Traversals Investigate

LESSON 10 [CODE.ORG]



Traversals Practice

LESSON 11[CODE.ORG]

Traversals Make

LESSON 12 [CODE.ORG]