1. UPDATING VARIABLES
   a. \( x = 0 \)
   b. \( x = x + 1 \)

2. ITERATIONS: WHILE LOOPS
   a. Unlike us, computers are particularly good at doing repetitive tasks.
   b. The \textit{while} statement tells the computer to repeat a block of statements until the condition is met.
   c. Example: Write a program that can do a \textit{countdown}.
   d. The basic format of a while statement is:
      i. Evaluate the condition, yielding true or false.
      ii. If the condition is false, it exits the \textit{while} statement.
      iii. If the condition is true, it executes the body and goes back to step i.
   e. You have to make sure the loop will terminate at some point. The variables in the condition have to change.
   f. Exercise: Figure out what this code does:
   g. Use of flags: variables used to know the state of a condition.

   ```
   acc = 0
   x = 4
   itersLeft = x
   while (itersLeft > 0):
       acc = acc + x
       itersLeft = itersLeft - 1
   print 'Total =', acc,'itersLeft=', itersLeft
   print acc
   ```

   h. Example: Write a program that asks the user a yes/no question and doesn’t quit until it receives a ‘yes’ or a ‘no.’ If it receives anything else, it asks the question again.

3. BREAK
   a. Sometimes you don’t know it’s time to end the loop until you get half way through the body.
   b. You can use the \textit{break} statement to force out of the loop.
   c. Example: Re-write the yes/no program using the break statements.

4. BRUTE FORCE ALGORITHMS
   a. Loops are often used in programs where you are trying to find an answer by enumerating all the possible solutions until you arrive at one that works.
   b. Example: Find all the divisors of a natural number \( n \). Enumerate all integers from 1 to \( n \), and check whether each of them divides \( n \) without a remainder.
   c. We can make the program a little better by stopping once we’ve reached \( n/2 \).
   d. Other examples of a brute force algorithm, a chess player that examines all possible arrangements.
   e. These algorithms tend to be easy to implement, and will find solutions if they exist, but it’s usually very costly. It takes a lot of time to run.
   f. Exercise: Write a program that finds the square root of a perfect square. (You can assume it only receives perfect squares as input).
   g. Theoretical examples: Traveling salesman and Knapsack problem. (We’ll come back to these later).

5. DEFENSIVE PROGRAMMING
   a. You can’t always rely that much on the user. It’s generally best to take some cases into consideration.
   b. What happens if you give it a negative number?
   c. Or what happens if the number is not a perfect square?
   d. Exercise: Re-write the program that finds the square root of a number with checks for those two conditions.
6. DATA STRUCTURES: LISTS
   a. A list is a sequence of values. The elements of the list can be of any type.
   b. The elements don’t have to be the same type.
   c. Lists can be nested within other lists.
   d. Lists can be empty.
   e. The syntax for accessing elements is `a[0]`. Elements start at 0 (and therefore end at n-1).
   f. Lists are mutable (you can change the elements as you go).
   g. There are a number of useful methods that you come built-in for lists.
      i. `len(a)`
      ii. `a.count(value1)`
      iii. `a.insert(position, value)`
      iv. `a.append(value)`
      v. `a.index(value)`
      vi. `a.remove(value)`
      vii. `a.reverse()`
      viii. `a.sort()`
   h. You can ask whether an element is in a list: `carlos in names`.
   i. Slicing: `a[0:]`
   j. Some methods allow to use a list as a stack: last in, first out.
      i. `b.append(6)`
      ii. `b.pop()`
   k. There are several automatic ways of making lists.
      i. `range()`
      ii. With the random library, you can easily shuffle a list, or sample it.
   l. There are also list operations.
      i. The + operator concatenates two lists.
      ii. The * operator repeats a list.

7. ITERATION: FOR LOOPS
   a. If I had a collection of elements, then I can simply walk through each of the items in the collection.
   b. Basic structure: for `<variable>` in `<some_collection>`: block of code
   c. Updates happen automatically.
   d. Exercise: Re-implement Find all divisors of X.
   e. As long as collection is finite, the program will terminate.
   f. What if we want to collect things together? A compound of elements?
   g. Say we now want to return all the divisors of x. So that another part of the program can use it.
   h. Traversing a list
      i. We can use a for loop to traverse a list. Show example with a list of names.

8. BE WARNED: COMPUTER SCIENTISTS LOVE OBFUSCATED SHORTCUTS
   a. Here are a few shorter ways of writing lists using for loops:
      i. `x = 10`    `divisors = [i for i in range(1,x) if x%i == 0]`
   b. Other interesting shortcuts:
      i. `squares = [x**2 for x in range(10)]`
      ii. `(x, y) for x in [1,2,3] for y in [3,1,4] if x != y`
   c. Knowing how to write functions, there are cool things you can do with lists.
      i. FILTER:
         1. def even(x): return x%2 == 0
         2. filter(even, range(1,10))
      ii. MAP:
         1. def cube(x): return x*x*x
         2. map(cube, range(1,10))

9. NESTED ITERATIONS
   a. You can write a while/for loop inside a while/for loop.
   b. Although it’s often useful to subdivide the problem, and write a separate function.
   c. Example: Suppose you want to find the divisors of a list of numbers.