7.2. The **for** Loop

A basic building block of all programs is to be able to repeat some code over and over again. We refer to this repetitive idea as **iteration**. In this section, we will explore some mechanisms for basic iteration.

In Python, the **for** statement allows us to write programs that implement iteration. As a simple example, let’s say we have some friends, and we’d like to send them each an email inviting them to our party. We don’t quite know how to send email yet, so for the moment we’ll just print a message for each friend.

for name in ["Joe", "Amy", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]:

print("Hi", name, "Please come to my party on Saturday!")

​

Take a look at the output produced when you press the run button. There is one line printed for each friend. Here’s how it works:

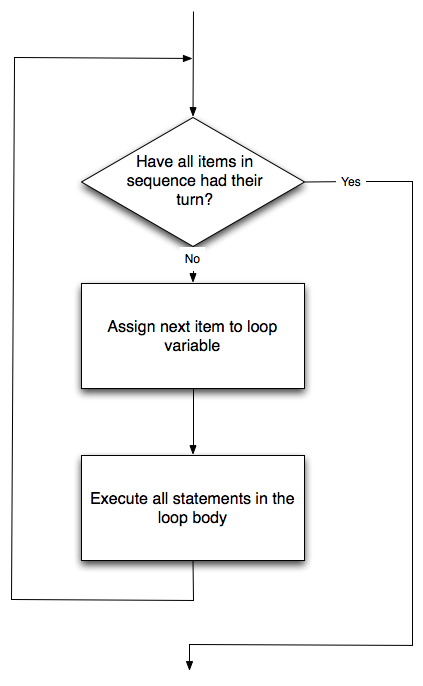
* **name** in this for statement is called the **loop variable**.
* The list of names in the square brackets is the sequence over which we will iterate.
* Line 2 is the **loop body**. The loop body is always indented. The indentation determines exactly what statements are “in the loop”. The loop body is performed one time for each name in the list.
* On each *iteration* or *pass* of the loop, first a check is done to see if there are still more items to be processed. If there are none left (this is called the **terminating condition** of the loop), the loop has finished. Program execution continues at the next statement after the loop body.
* If there are items still to be processed, the loop variable is updated to refer to the next item in the list. This means, in this case, that the loop body is executed here 7 times, and each time name will refer to a different friend.
* At the end of each execution of the body of the loop, Python returns to the for statement, to see if there are more items to be handled.

# 7.3. Flow of Execution of the for Loop

As a program executes, the interpreter always keeps track of which statement is about to be executed. We call this the **control flow**, or the **flow of execution** of the program. When humans execute programs, they often use their finger to point to each statement in turn. So you could think of control flow as “Python’s moving finger”.

Control flow until now has been strictly top to bottom, one statement at a time. We call this type of control **sequential**. Sequential flow of control is always assumed to be the default behavior for a computer program. The for statement changes this.

Flow of control is often easy to visualize and understand if we draw a flowchart. This flowchart shows the exact steps and logic of how the for statement executes.

[](https://fopp.umsi.education/runestone/static/fopp/_images/new_flowchart_for.png)

# 7.4. Strings and for loops

Since a string is simply a sequence of characters, the for loop iterates over each character automatically. (As always, try to predict what the output will be from this code before your run it.)

for achar in "Go Spot Go":

print(achar)

​

The loop variable achar is automatically reassigned each character in the string “Go Spot Go”. We will refer to this type of sequence iteration as **iteration by item**. Note that the for loop processes the characters in a string or items in a sequence one at a time from left to right.

**Check your understanding**

iter-4-1: How many times is the word HELLO printed by the following statements?

s = "python rocks"

**for** ch **in** s:

**print**("HELLO")

Top of Form

A. 10  
B. 11  
C. 12  
D. Error, the for statement needs to use the range function.

Bottom of Form

iter-4-2: How many times is the word HELLO printed by the following statements?

s = "python rocks"

**for** ch **in** s[3:8]:

**print**("HELLO")

Top of Form

A. 4  
B. 5  
C. 6  
D. Error, the for statement cannot use slice.

# 7.5. Lists and for loops

It is also possible to perform **list traversal** using iteration by item. A list is a sequence of items, so the for loop iterates over each item in the list automatically.

fruits = ["apple","orange","banana","cherry"]

​

for afruit in fruits: # by item

print(afruit)

​

It almost reads like natural language: For (every) fruit in (the list of) fruits, print (the name of the) fruit.

7.6. The Accumulator Pattern

One common programming “pattern” is to traverse a sequence, **accumulating** a value as we go, such as the sum-so-far or the maximum-so-far. That way, at the end of the traversal we have accumulated a single value, such as the sum total of all the items or the largest item.

**The anatomy of the accumulation pattern includes:**

* **initializing** an “accumulator” variable to an initial value (such as 0 if accumulating a sum)
* **iterating** (e.g., traversing the items in a sequence)
* **updating** the accumulator variable on each iteration (i.e., when processing each item in the sequence)

For example, consider the following code, which computes the sum of the numbers in a list.

nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

or w in nums:

accum = accum + w

print(accum)

​

In the program above, notice that the variable accum starts out with a value of 0. Next, the iteration is performed 10 times. Inside the for loop, the update occurs. w has the value of current item (1 the first time, then 2, then 3, etc.). accum is reassigned a new value which is the old value plus the current value of w.

This pattern of iterating the updating of a variable is commonly referred to as the **accumulator pattern**. We refer to the variable as the **accumulator**. This pattern will come up over and over again. Remember that the key to making it work successfully is to be sure to initialize the variable before you start the iteration. Once inside the iteration, it is required that you update the accumulator.

**Check your understanding**

iter-6-1: Consider the following code:

nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

**for** w **in** nums:

accum = 0

accum = accum + w

**print**(accum)

What happens if you put the initialization of accum inside the for loop as the first instruction in the loop?

Top of Form

A. It will print out 10 instead of 55  
B. It will cause a run-time error  
C. It will print out 0 instead of 55

# 7.7. Traversal and the for Loop: By Index

It is also possible to iterate through the indexes of a string or sequence. The for loop can then be used to iterate over these positions. These positions can be used together with the indexing operator to access the individual characters in the string. We can use **Enumerate**, a built-in Python function, to make this proccess easier because it allows us to loop through something and have an automatic counter.

for counter,item in enumerate(['apple','pear','apricot','cherry','peach’]):

print(counter, item)

​

By using the enumerate function, we can print out a counter that tells us the position of an item in a list. We could do this ourselves, but this saves us from having to do that. The index positions in the list are 0, 1 , 2, 3, and 4. This is exactly the same sequence of integers that are stored in counter each time the loop is iterated. The first time through the for loop, counter will be 0 and “apple” will be printed. Then, counter will be reassigned to 1 and “pear” will be displayed. This will continue until the list has ended, so that the final value for counter will be 4 and the final value of item will be “peach”.

Conveniently, we can also use the range function to automatically generate the indices of the characters.

x = range(5)

print(type(x))

print(x)

In order to make the iteration more general, we can use the len function to provide the bound for range. This is a very common pattern for traversing any sequence by position. Make sure you understand why the range function behaves correctly when using len of the string as its parameter value.

fruit = ['apple', 'pear', 'apricot', 'cherry', 'peach']

for n in range(len(fruit)):

print(n, fruit[n])

**Check your understanding**

moreiter-6-1: How many times is the letter p printed by the following statements?

s = "python"

**for** idx **in** range(len(s)):

**print**(s[idx % 2])

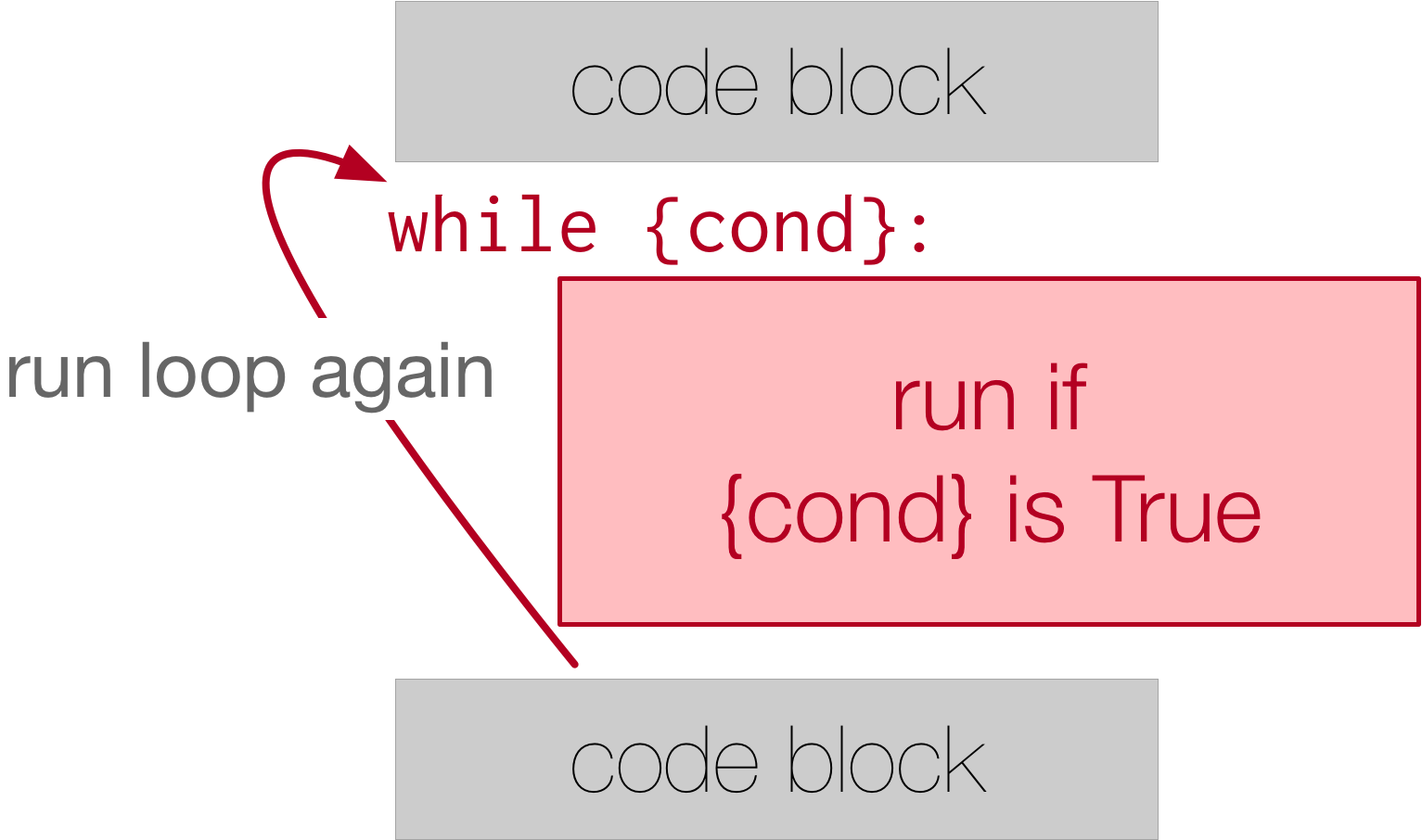
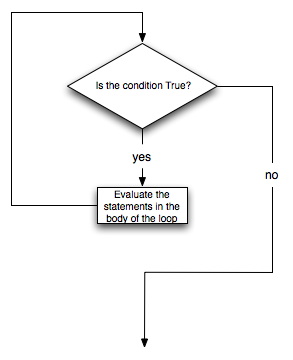
Top of Form

A. 0  
B. 1  
C. 2  
D. 3  
E. 6

# 7.8. The while Statement

There is another Python statement that can also be used to build an iteration. It is called the while statement. The while statement provides a much more general mechanism for iterating. Similar to the if statement, it uses a boolean expression to control the flow of execution. The body of while will be repeated as long as the controlling boolean expression evaluates to True.

The following two figures show the flow of control. The first focuses on the flow inside the while loop and the second shows the while loop in context.



We can use the while loop to create any type of iteration we wish, including anything that we have previously done with a for loop. For example, the program in the previous section could be rewritten using while. Instead of relying on the range function to produce the numbers for our summation, we will need to produce them ourselves. To do this, we will create a variable called aNumber and initialize it to 1, the first number in the summation. Every iteration will add aNumber to the running total until all the values have been used. In order to control the iteration, we must create a boolean expression that evaluates to True as long as we want to keep adding values to our running total. In this case, as long as aNumber is less than or equal to the bound, we should keep going.

Here is a new version of the summation program that uses a while statement.

def sumTo(aBound):

""" Return the sum of 1+2+3 ... n """

​

theSum = 0

aNumber = 1

while aNumber <= aBound:

theSum = theSum + aNumber

aNumber = aNumber + 1

return theSum

​

print(sumTo(4))

​

print(sumTo(1000))

​

You can almost read the while statement as if it were in natural language. It means, while aNumber is less than or equal to aBound, continue executing the body of the loop. Within the body, each time, update theSum using the accumulator pattern and increment aNumber. After the body of the loop, we go back up to the condition of the while and reevaluate it. When aNumber becomes greater than aBound, the condition fails and flow of control continues to the return statement.

One very common pattern is called a **listener loop**. Inside the while loop there is a function call to get user input. The loop repeats indefinitely, until a particular input is received.

theSum = 0

x = -1

while (x != 0):

x = int(input("next number to add up (enter 0 if no more numbers): "))

theSum = theSum + x

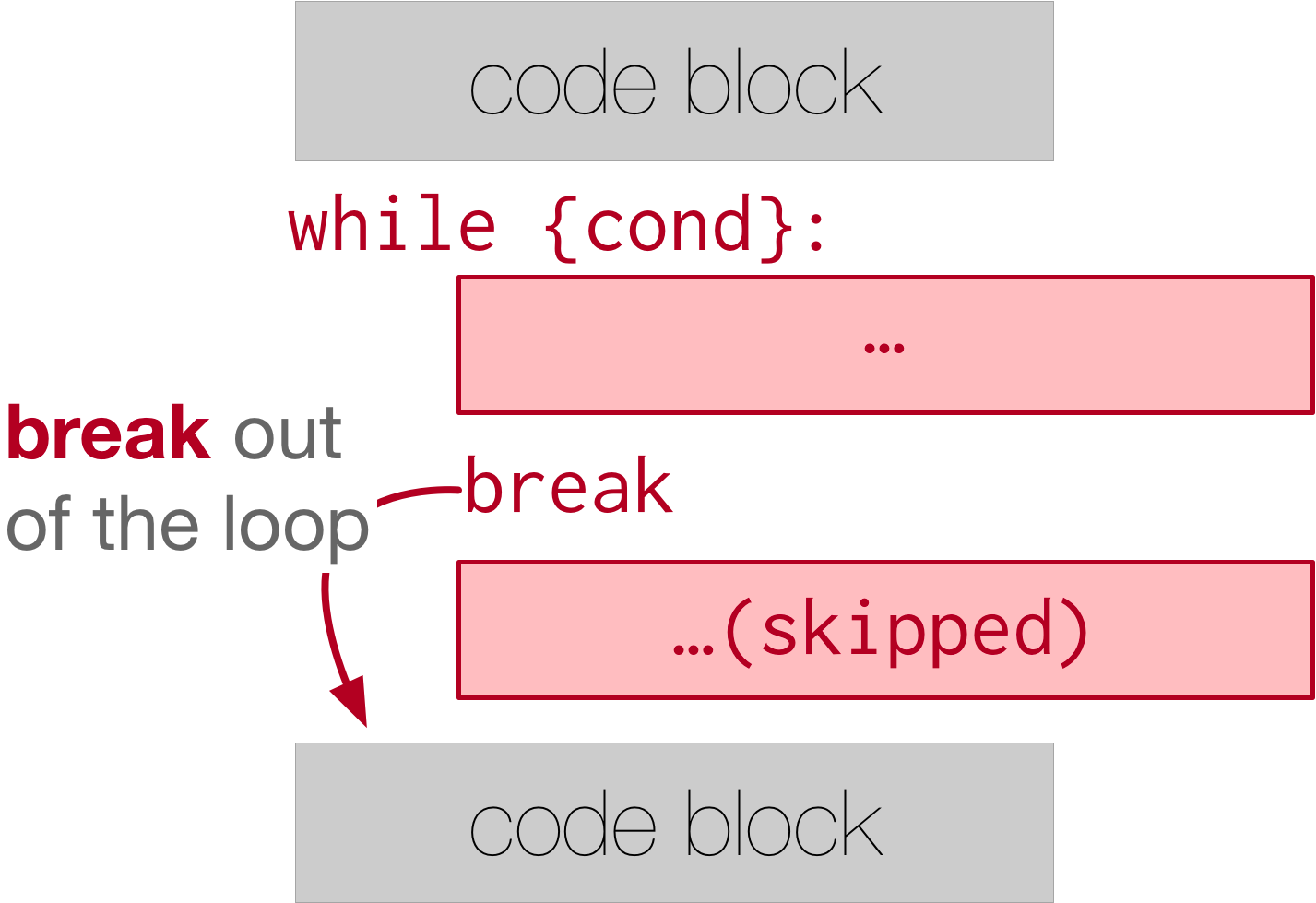
​

print(theSum)

# 7.9. Break and Continue

Python provides ways for us to control the flow of iteration with a two keywords: break and continue.

break allows the program to immediately ‘break out’ of the loop, regardless of the loop’s conditional structure. This means that the program will then skip the rest of the iteration, without rechecking the condition, and just goes on to the next outdented code that exists after the whole while loop.



while True:

print("this phrase will always print")

break

print("Does this phrase print?")

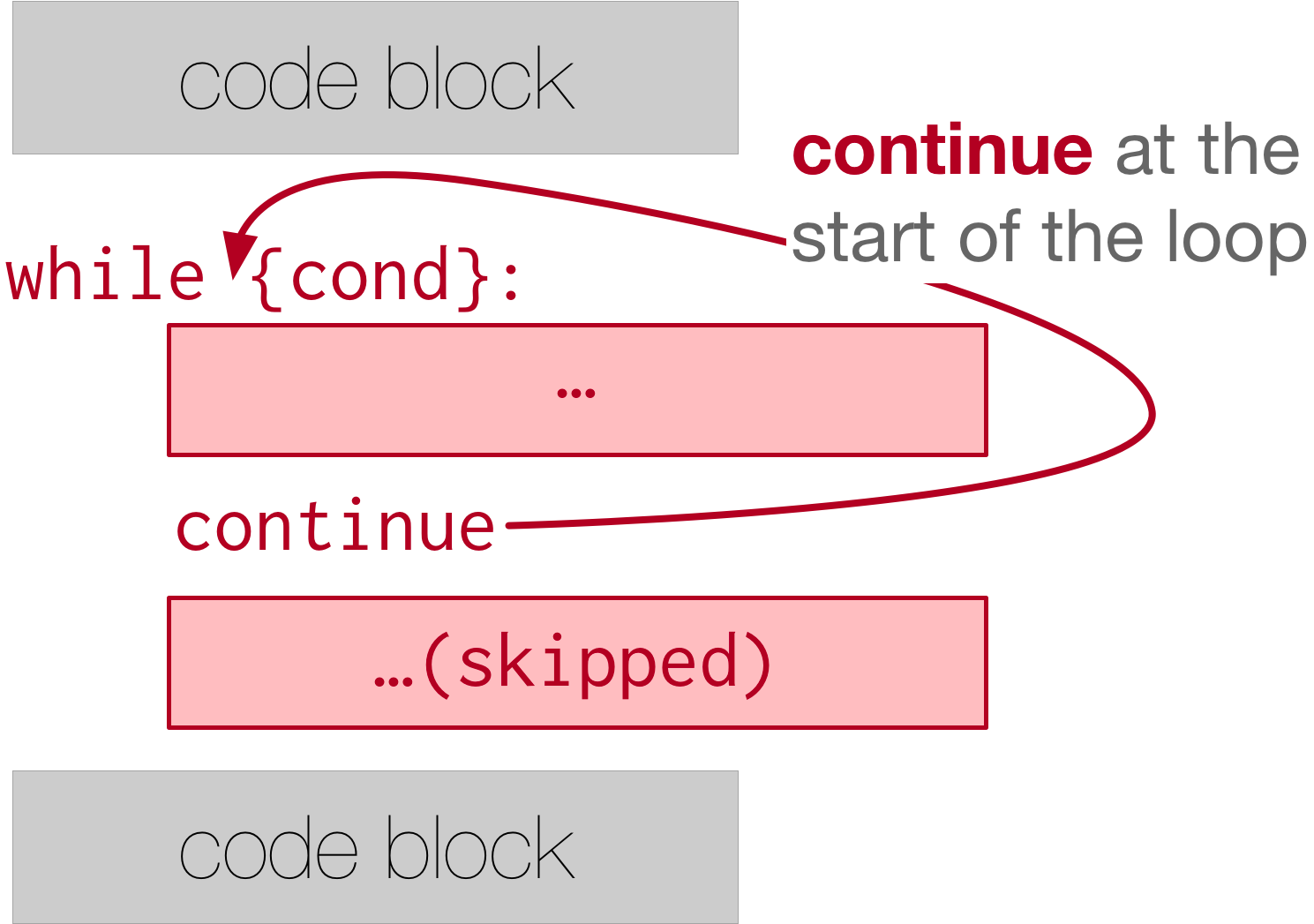
​

print("We are done with the while loop.")

​

We can see here how the print statement right after break is not executed. In fact, without using break, we have no way to stop the while loop because the condition is always set to True!

continue is the other keyword that can control the flow of iteration. Using continue allows the program to immediately “continue” with the next iteration. The program will skip the rest of the iteration, recheck the condition, and maybe does another iteration depending on the condition set for the while loop.



x = 0

while x < 10:

print("we are incrementing x")

if x % 2 == 0:

x += 3

continue

if x % 3 == 0:

x += 5

x += 1

print("Done with our loop! X has the value: " + str(x))

# 7.10. Glossary

**for loop traversal (for)**

Traversing a string or a list means accessing each character in the string or item in the list, one at a time. For example, the following for loop:

**for** ix **in** 'Example':

...

executes the body of the loop 7 times with different values of ix each time.

**range**

A function that produces a list of numbers. For example, range(5), produces a list of five numbers, starting with 0, [0, 1, 2, 3, 4].

**pattern**

A sequence of statements, or a style of coding something that has general applicability in a number of different situations. Part of becoming a mature programmer is to learn and establish the patterns and algorithms that form your toolkit.

**index**

A variable or value used to select a member of an ordered collection, such as a character from a string, or an element from a list.

**traverse**

To iterate through the elements of a collection, performing a similar operation on each.

**accumulator pattern**

A pattern where the program initializes an accumulator variable and then changes it during each iteration, accumulating a final result.

Bottom of Form

Bottom of Form

Bottom of Form