INT3075 Programming and Problem Solving for Mathematics

Control (Part II): Repetition

Repetition, quick overview

Repeating statements

- Besides selecting which statements to execute, a fundamental need in a program is repetition
  - repeat a set of statements under some conditions
- With both selection and repetition, we have the two most necessary programming statements

While and For statements

- The `while` statement is the more general repetition construct. It repeats a set of statements while some condition is True.
- The `for` statement is useful for iteration, moving through all the elements of data structure, one at a time.

`while` loop

- Top-tested loop (pretest)
  - test the boolean before running
  - test the boolean before each iteration of the loop

```
while boolean expression:
    suite
```

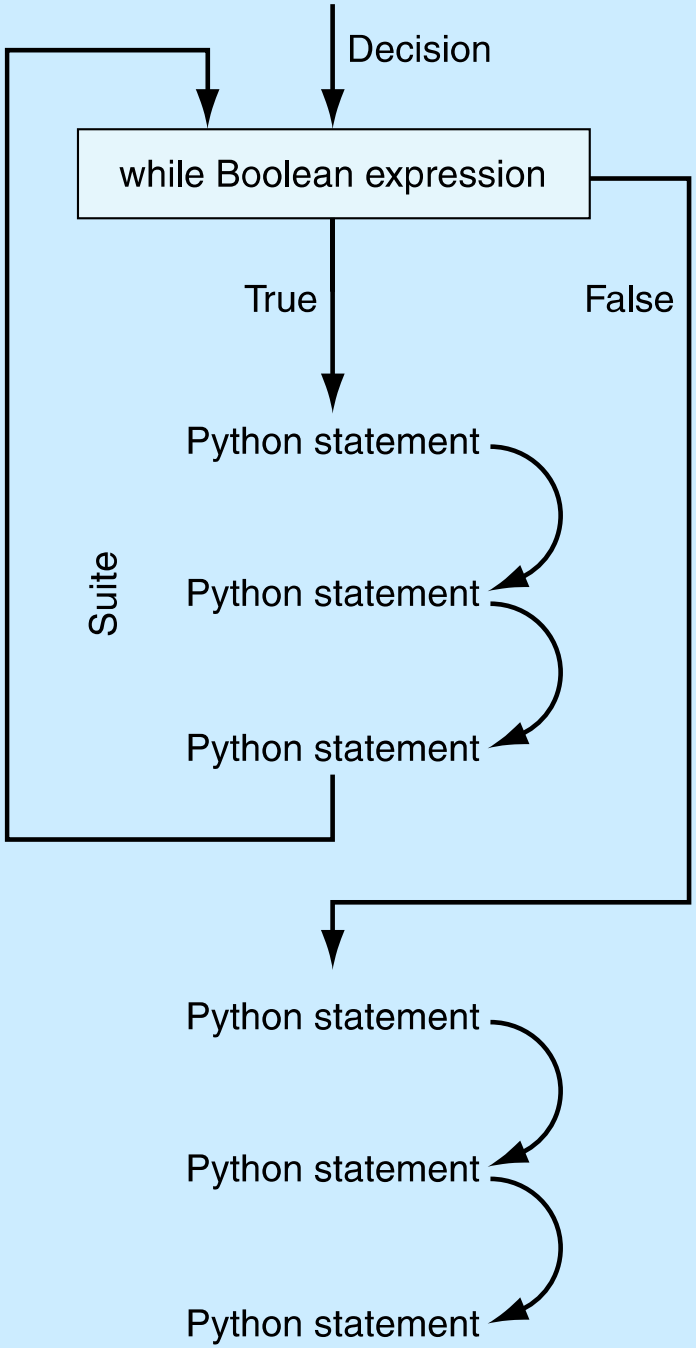
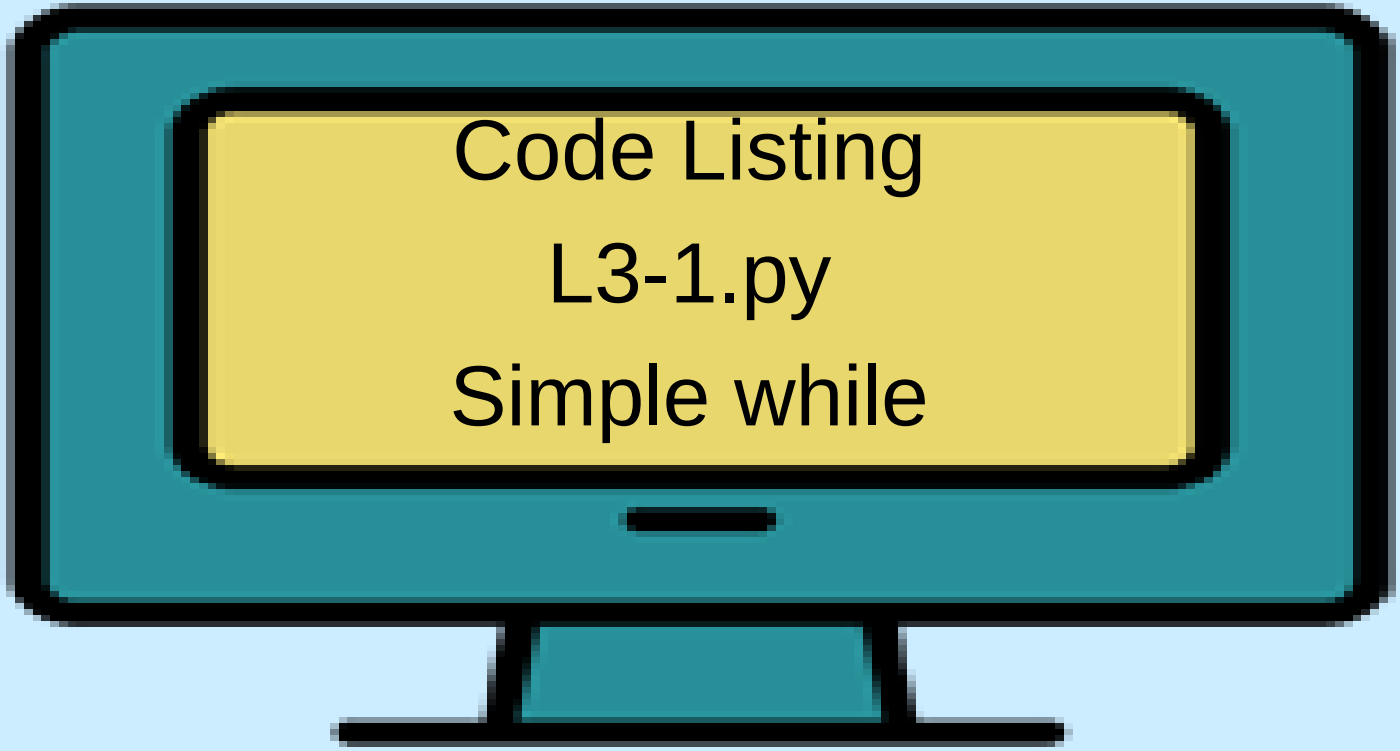


FIGURE 2.4 ***`while`*** loop.

repeat while the boolean is true

- while loop will repeat the statements in the suite while the boolean is `True` (or its Python equivalent)
- If the Boolean expression never changes during the course of the loop, the loop will continue forever.


Code Listing
L3-1.py
Simple while

```
# simple while

x_int = 0      # initialize loop-control variable

# test loop-control variable at beginning of loop
while x_int < 10:
    print(x_int, end=' ')  # print the value of x_int each time through the
                             while loop
    x_int = x_int + 1      # change loop-control variable

print()
print("Final value of x_int: ", x_int)  # bigger than value printed in loop!
```

General approach to a while

- outside the loop, initialize the boolean
- somewhere inside the loop you perform some operation which changes the state of the program, eventually leading to a False boolean and exiting the loop
- Have to have both!

for and iteration

- One of Python's strength's is it's rich set of built-in data structures
- The `for` statement iterates through each element of a collection (list, etc.)

```
for element in collection:
    suite
```

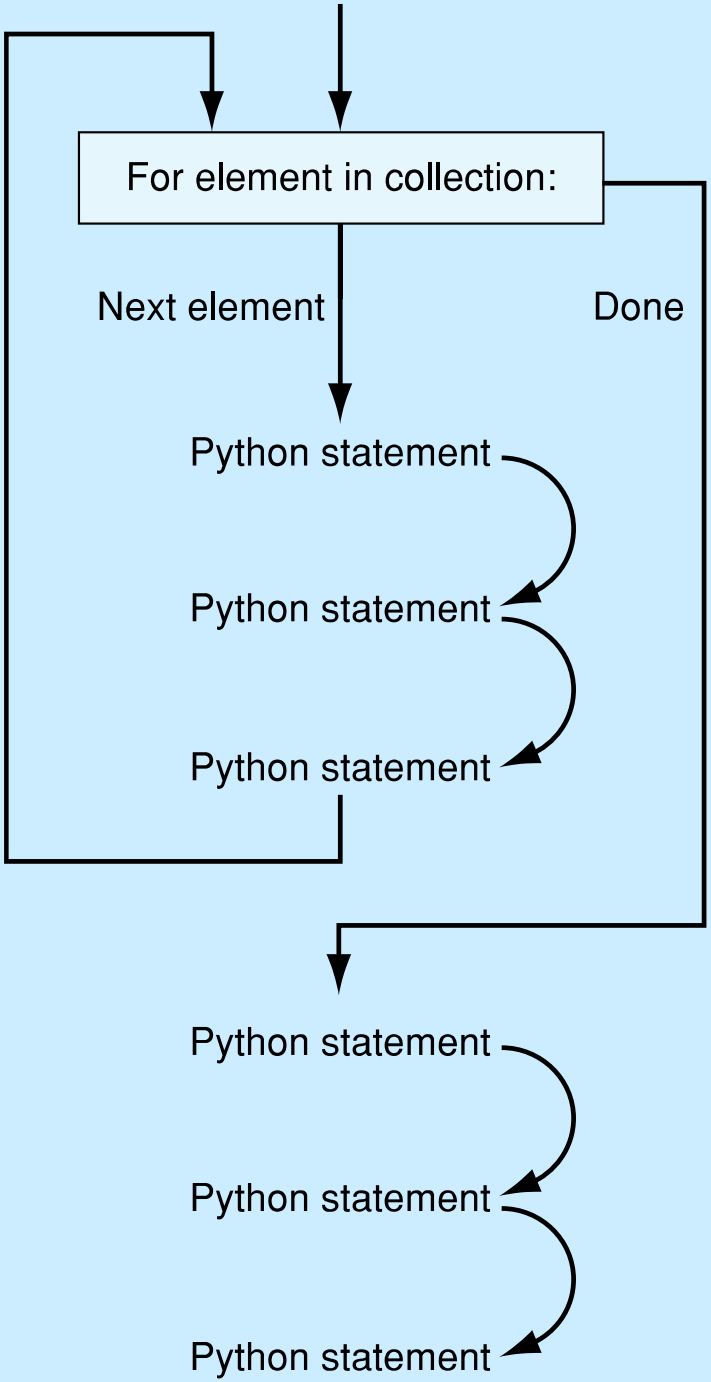
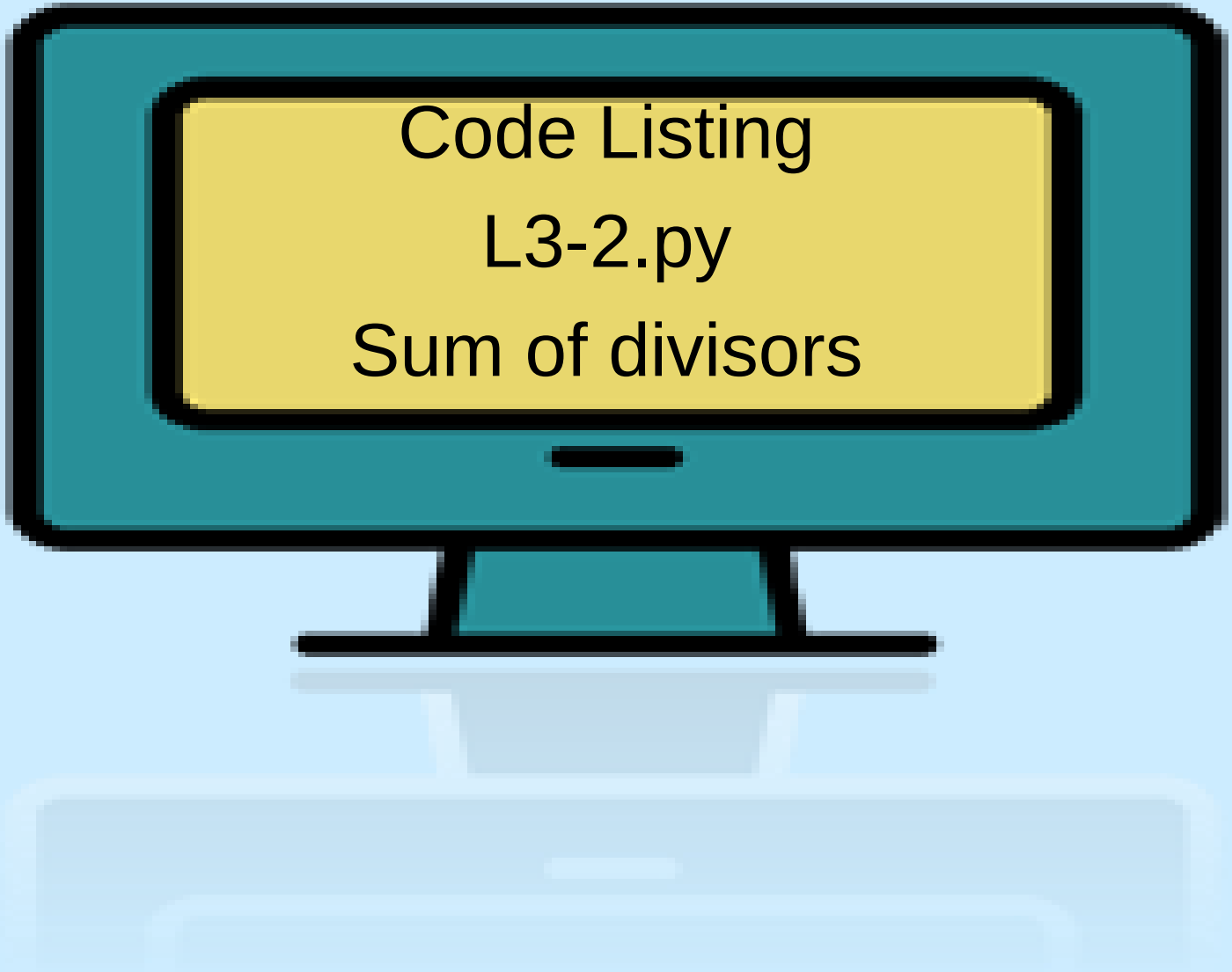


FIGURE 2.5 Operation of a ***for*** loop.


Code Listing
L3-2.py
Sum of divisors

Calculate Sum of Divisors

```
divisor = 1
sum_of_divisors = 0
while divisor < number:
    if number % divisor == 0:        # divisor evenly divides theNum
        sum_of_divisors = sum_of_divisors + divisor
    divisor = divisor + 1
```

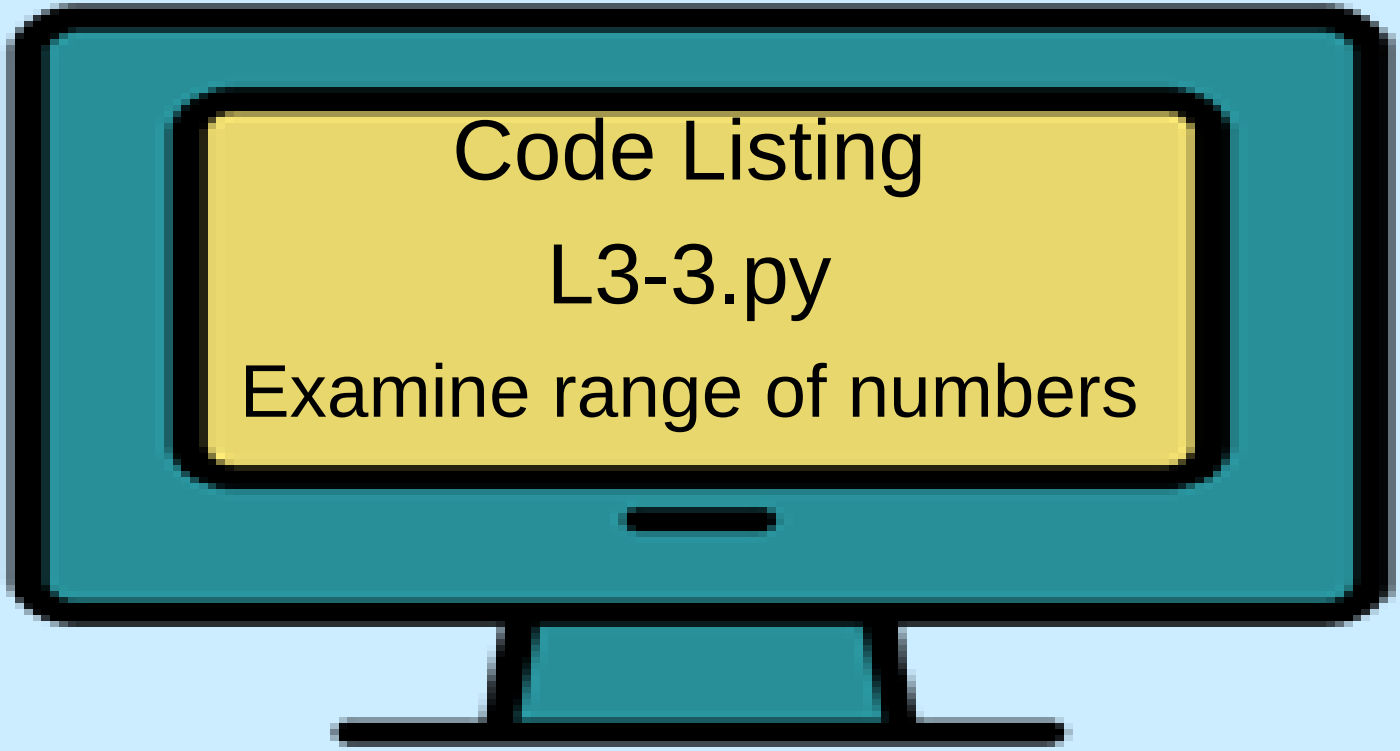
Improving the Perfect Number Program

Work with a range of numbers

For each number in the range of numbers:

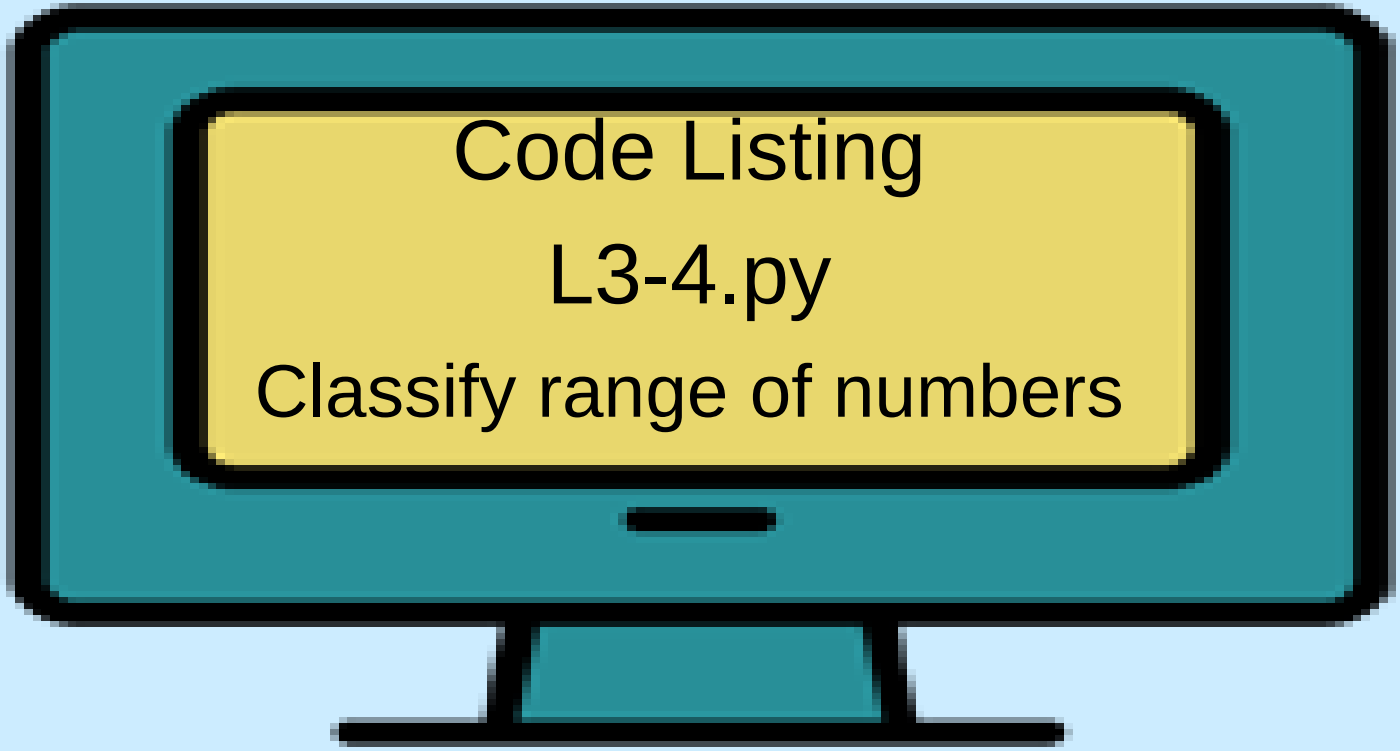
- collect all the factors
- once collected, sum up the factors
- compare the sum and the number and respond accordingly

Print a summary


Code Listing
L3-3.py
Examine range of numbers

Examine a range of numbers

```
top_num_str = input("What is the upper number for the range:")
top_num = int(top_num_str)
number=2
while number <= top_num:
    # sum the divisors of number
    # classify the number based on its divisor sum
    number += 1
```


Code Listing
L3-4.py
Classify range of numbers

Classify range of numbers

```
# classify a range of numbers with respect to perfect, adundant or deficient
# unless otherwise stated, variables are assumed to be of type int. Rule 4

top_num_str = input("What is the upper number for the range:")
top_num = int(top_num_str)
number=2
while number <= top_num:
    # sum up the divisors
    divisor = 1
    sum_of_divisors = 0
    while divisor < number:
        if number % divisor == 0:

    # classify the number based on its divisor sum
    if number == sum_of_divisors:
        print(number,"is perfect")
    if number < sum_of_divisors:
        print(number,"is abundant")
    if number > sum_of_divisors:
        print(number,"is deficient")
    number += 1
```

More Control: Repetition

Developing a while loop

Working with the ***loop control variable***:

- Initialize the variable, typically outside of the loop and before the loop begins.
- The condition statement of the while loop involves a Boolean using the variable.
- Modify the value of the control variable during the course of the loop

Issues

Loop never starts:

- the control variable is not initialized as you thought (or perhaps you don't always want it to start)

Loop never ends:

- the control variable is not modified during the loop (or not modified in a way to make the Boolean come out `False`)

while loop, round two

- while loop, oddly, can have an associated `else` suite
- `else` suite is executed when the loop finishes under normal conditions
  - basically the last thing the loop does as it exits

while with else

```
while booleanExpression:
  suite
  suite
else:
  suite
  suite
rest of the program
```

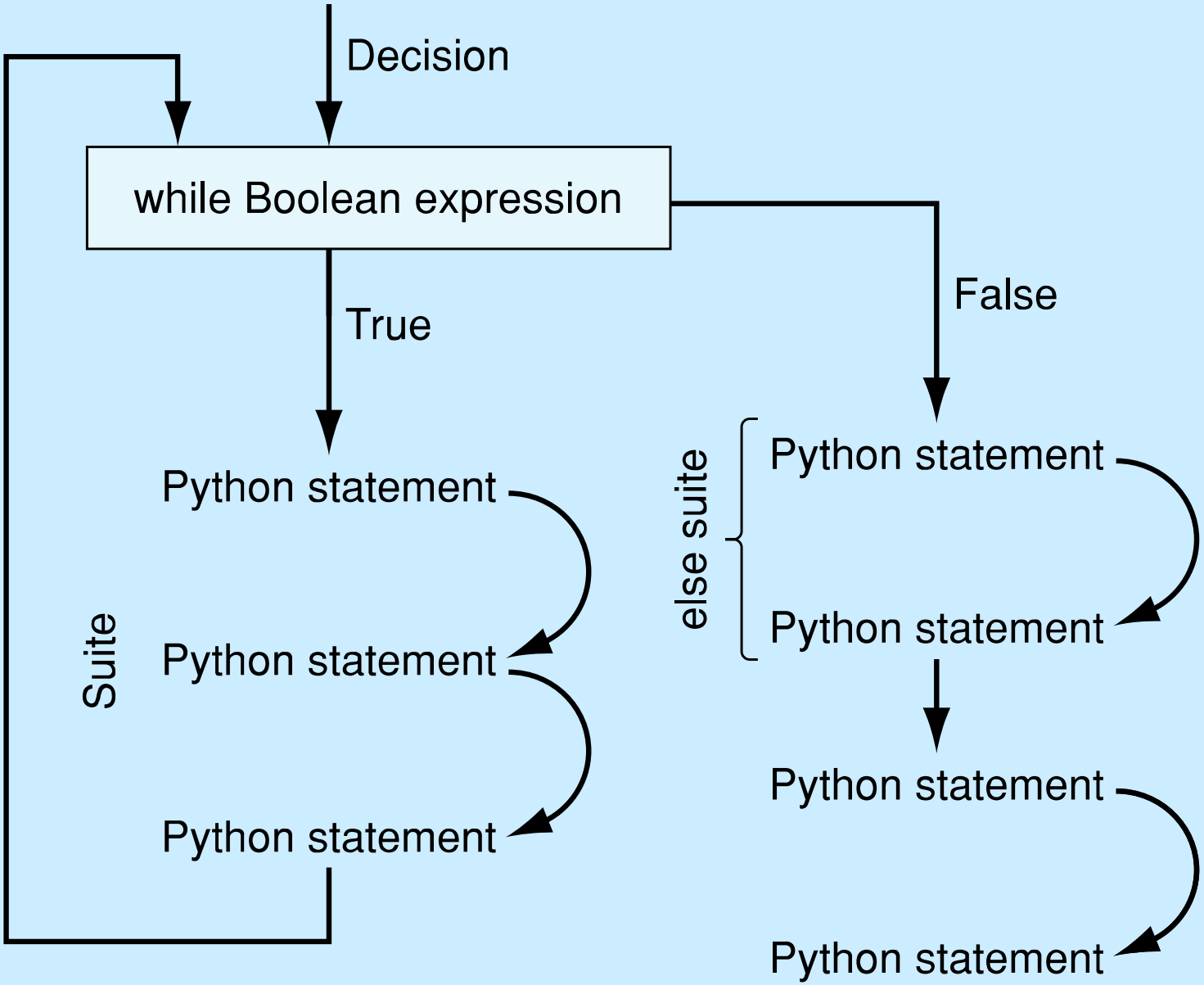
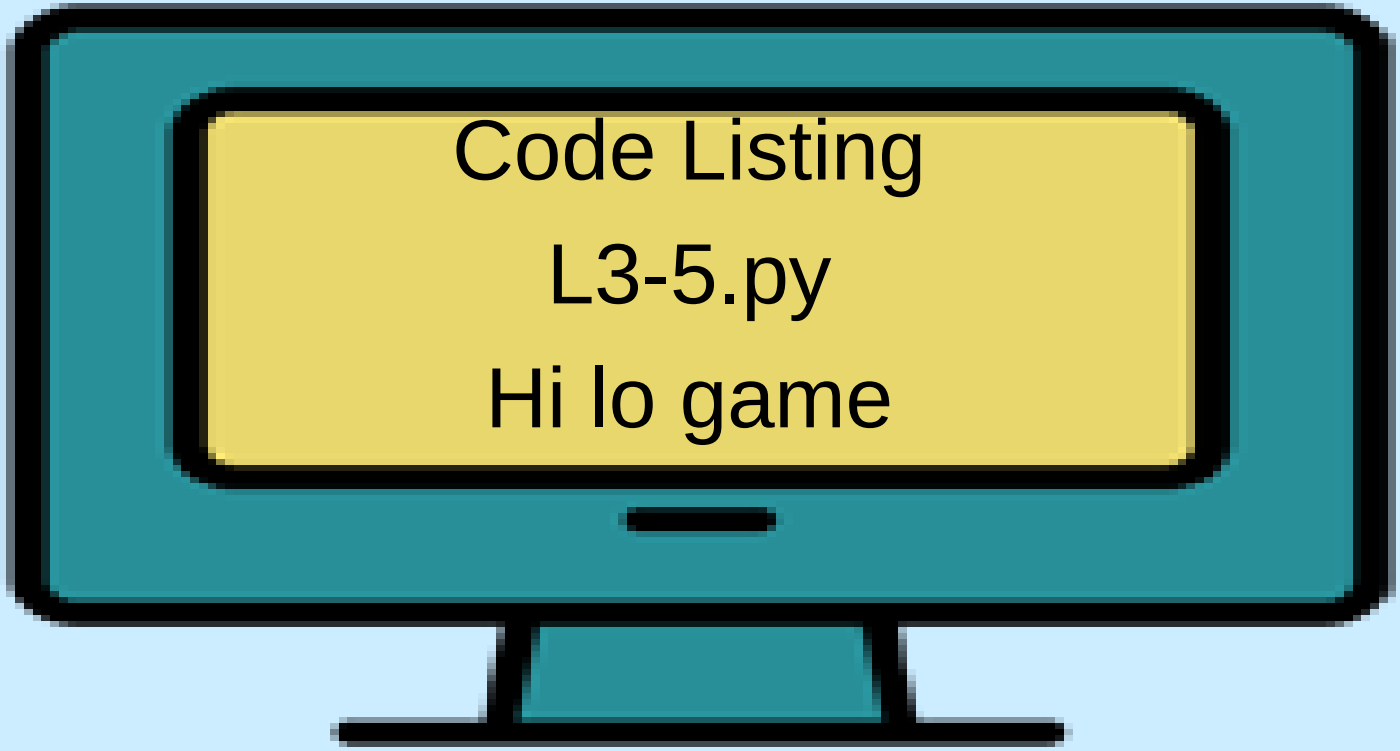


FIGURE 2.9 ***while-else.***

Break statement

- A `break` statement in a loop, if executed, exits the loop
- It exists immediately, skipping whatever remains of the loop as well as the else statement (if it exists) of the loop


Code Listing
L3-5.py
Hi lo game

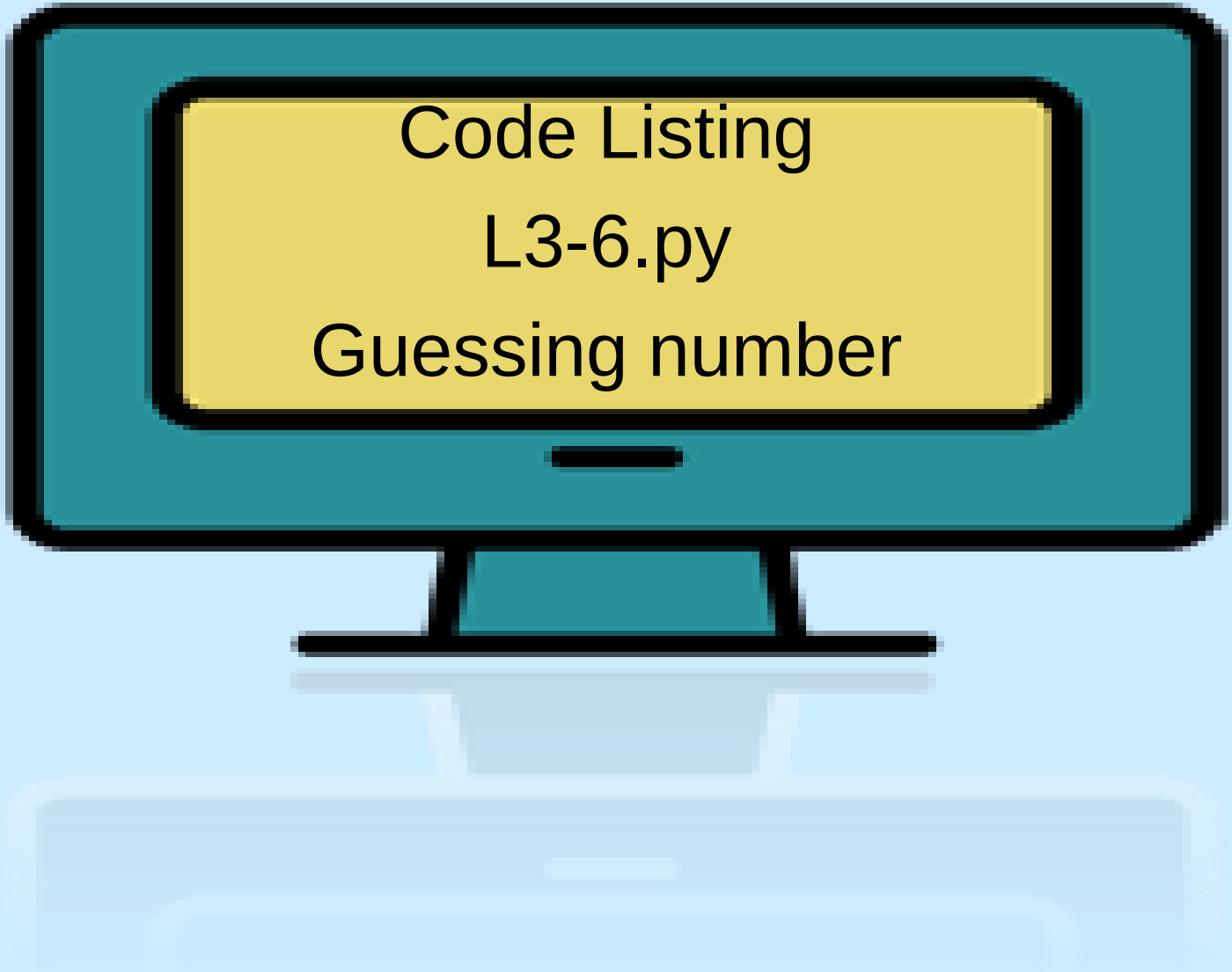
Hi Lo Game

```
# get an initial guess
guess_str = input("Guess a number: ")
guess = int(guess_str)  # convert string to number

# while guess is range, keep asking
while 0 <= guess <= 100:
    if guess > number:
        print("Guessed Too High.")
    elif guess < number:
        print("Guessed Too Low.")
    else:                   # correct guess, exit with break
        print("You guessed it. The number was:",number)
        break
    # keep going, get the next guess
    guess_str = input("Guess a number: ")
    guess = int(guess_str)
else:
    print("You quit early, the number was:",number)
```

Continue statement

- A `continue` statement, if executed in a loop, means to immediately jump back to the top of the loop and re-evaluate the conditional
- Any remaining parts of the loop are skipped for the one iteration when the continue was executed


Code Listing
L3-6.py
Guessing number

Part of the guessing numbers program

```
# initialize the input number and the sum
number_str = input("Number: ")
the_sum = 0

# Stop if a period (.) is entered.
# remember, number_str is a string  until we convert it
while number_str != "." :
    number = int(number_str)
    if number % 2 == 1:  # number is not even (it is odd)
        print ("Error, only even numbers please.")
        number_str = input("Number: ")
        continue     # if the number is not even, ignore it
    the_sum += number
    number_str = input("Number: ")

print ("The sum is:",the_sum)
```

change in control: Break and Continue

- while loops are easiest read when the conditions of exit are clear
- Excessive use of continue and break within a loop suite make it more difficult to decide when the loop will exit and what parts of the suite will be executed each loop.

While overview

```
while test1:
    statement_list_1
    if test2:  break          # Exit loop now; skip else
    if test3:  continue       # Go to top of loop now
    # more statements
else:
    statement_list_2          # If we didn't hit a 'break'

# 'break' or 'continue' lines can appear anywhere
```

Range and for loop

Range function

- The range function represents a sequence of integers
- the range function takes 3 arguments:
  - the beginning of the range. Assumed to be 0 if not provided
  - the end of the range, but not inclusive (up to but not including the number). ***Required***
  - the step of the range. Assumed to be 1 if not provided
- if only one argument is provided, it is assumed to be the end value

Iterating through the sequence

```
for num in range(1,5):
    print(num)
```

- range represents the sequence 1, 2, 3, 4
- for loop assigns `num` to each of the values in the sequence, one at a time, in sequence
- prints each number (one number per line)

range generates on demand

Range generates its values on demand

```
>>> range(1,10)
range(1, 10)
>>> my_range=range(1,10)
>>> type(my_range)
<class 'range'>
>>> len(my_range)
9
>>> for i in my_range:
        print(i, end=' ')

1 2 3 4 5 6 7 8 9
>>>
```

Hailstone example

Hailstone sequence

- The Hailstone sequence is a simple algorithm applied to any positive integer
- In general, by applying this algorithm to your starting number you generate a sequence of other positive numbers, ending at 1
- Sequences go up and down just like a hailstone in a cloud

Algorithm

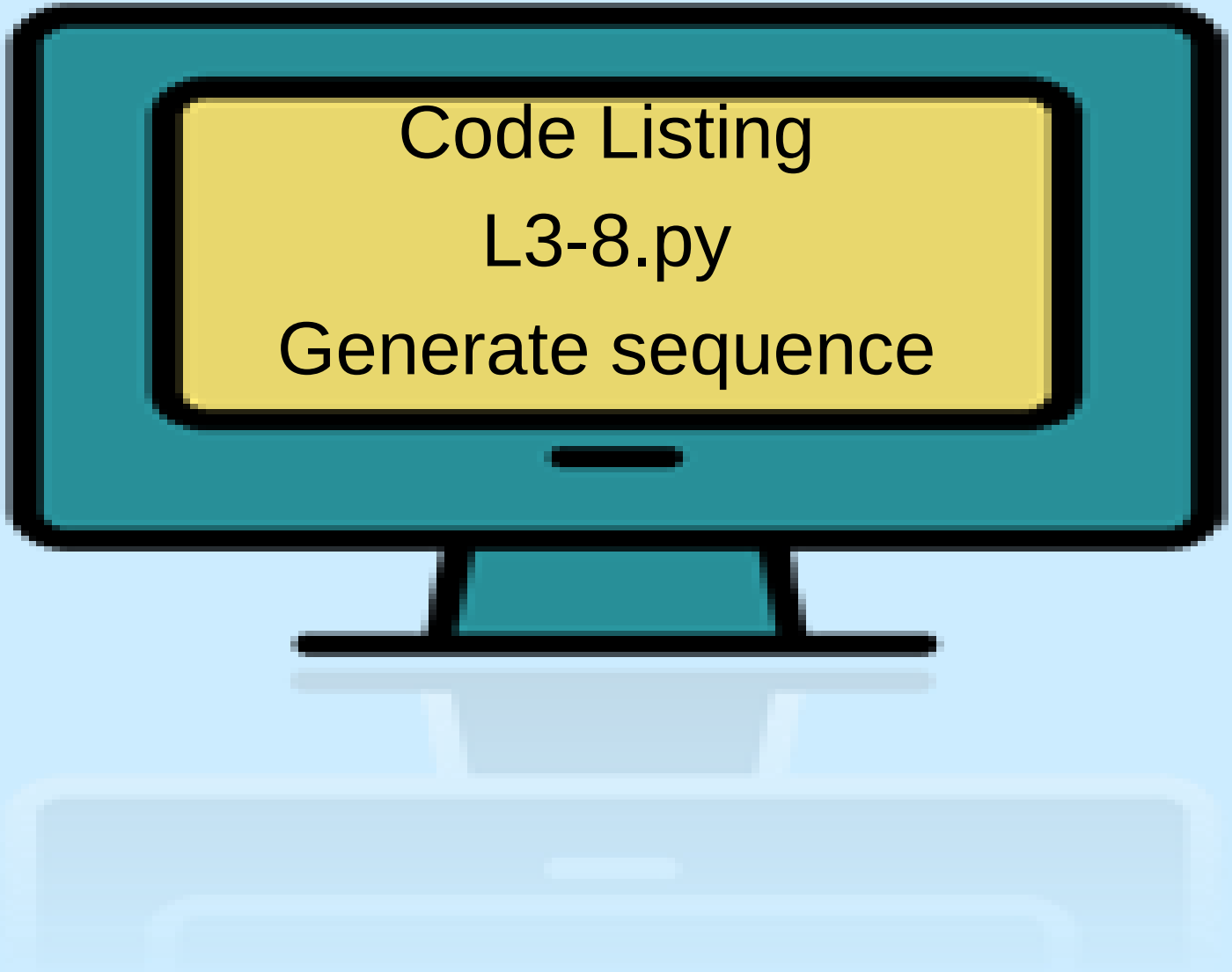
while the number does not equal one

- If the number is odd, multiply by 3 and add 1
- If the number is even, divide by 2
- Use the new number and reapply the algorithm

Even and Odd

Use the remainder operator

- `if num % 2 == 0:` # even
- `if num % 2 == 1:` # odd
- `if num %2:` # odd (why???)


Code Listing
L3-8.py
Generate sequence

```python
# Generate a hailstone sequence
number_str = input("Enter a positive integer:")
number = int(number_str)
count = 0

print("Starting with number:",number)
print("Sequence is: ", end=' ')

while number > 1:  # stop when the sequence reaches 1

    if number%2:          # number is odd
        number = number*3 + 1
    else:               # number is even
        number = number/2
    print(number,",", end=' ')    # add number to sequence

    count +=1          # add to the count

else:
    print()     # blank line for nicer output
    print("Sequence is ",count," numbers long")
```