Repeating Code Using Loops

There are three concepts for any language:

- Sequential
- Selection
- Repetition

Introduction of fundamental kind of control flow:

repetition.

Up to now, to execute an instruction two hundred times, you would need to write that instruction two hundred times. Now you'll see how to write the instruction once and use loops to repeat that code the desired number of times.

Processing Items in a List

With what you've learned so far, to print the items from a list of velocities of falling objects in metric, you would need to write a call on function *print* for each velocity in the list:

```
>>> velocities = [0.0, 9.81, 19.62, 29.43]
>>> print('Metric:', velocities[0], 'm/sec;',
... 'Imperial:', velocities[0] * 3.28, 'ft/sec')
Metric: 0.0 m/sec; Imperial: 0.0 ft/sec
>>> print('Metric:', velocities[1], 'm/sec;',
... 'Imperial:', velocities[1] * 3.28, 'ft/sec')
Metric: 9.81 m/sec; Imperial: 32.1768 ft/sec
>>> print('Metric:', velocities[2], 'm/sec; ',
... 'Imperial:', velocities[2] * 3.28, 'ft/sec')
Metric: 19.62 m/sec; Imperial: 64.3536 ft/sec
>>> print('Metric:', velocities[3], 'm/sec; ',
... 'Imperial:', velocities[3] * 3.28, 'ft/sec')
Metric: 29.43 m/sec; Imperial: 96.5304 ft/sec
```

Processing Items in a List

- The code above is used to process a list with just four values.
- Just imagine if you want to create report of 1000's of values
- Is it practical to repeat print 1000's times

Python has a *for loop* is the solution

```
>>> velocities = [0.0, 9.81, 19.62, 29.43]
>>> for velocity in velocities:
... print('Metric:', velocity, 'm/sec;',
... 'Imperial:', velocity * 3.28, 'ft/sec')
```

The general form of a for loop over a list is as follows:

for *«variable»* **in** *«list»*:
«block»

Execution of the loop

A for loop is executed as follows:

• The loop variable is assigned the first item in the list, and the loop block—the *body* of the for loop—is executed.

• The loop variable is then assigned the second item in the list and the loop body is executed again.

...

• Finally, the loop variable is assigned the last item of the list and the loop body is executed one last time.

• The loop stop when encounter empty item

Processing Characters in Strings

loop could go over the characters of a string.
for *«variable»* **in** *«str»*:
«block»

Ex:

```
def for_str():
  country = 'United State of America'
  for ch in country:
     if ch.isupper():
        print(ch)
```

In the code above, variable ch is assigned country[0] before the first iteration, country[1] before the second, and so on. The loop iterates twenty-four times (once per character) and the if statement block is checked 24 times and executed three times (once per uppercase letter).

Quiz: how can you print the output in one line:

Processing Characters in Strings

Example2:

find the vowel letters (aeiou) in a string

```
def print_vowels(word):
    for char in word:
        if char in 'aeiou':
            print(char)
```

Quiz: How can we make the solution perfect
Check both capital and small letters?

Looping Over a Range of Numbers

Create a range of values.
Loop over the range
This allows us to perform tasks a certain number of times

To begin, we need to generate the range of numbers over which to iterate.
Generating Ranges of Numbers:

```
>>> range(10)
```

Python's built-in function ***range*** produces an object that will generate a sequence of integers from 0 to 9.

Use a loop to access each number in the sequence one at a time:

```
>>> for num in range(10):
... print(num)
```

Quiz: Print even number of the above range

...

Looping Over a Range of Numbers

To get the numbers from the sequence all at once, we can use built-in function list to create a list of those numbers:

>>> list(range(10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Here are some more examples:

>>> list(range(3))

?

>>> list(range(1))

?

>>> list(range(0))

?

Function range can also be passed two arguments, where the first is the start value and the second is the stop value:

>>> list(range(1, 5)) **?**

>>> list(range(1, 10)) **?**

>>> list(range(5, 10)) **?**

Range with step size

By default, function range generates numbers that successively increase by one— this is called its *step size*.

We can specify a different step size for range with an optional third parameter:

>>> list(range(2000, 2050, 4)) ?

The step size can also be negative, which produces a descending sequence.
When the step size is negative, the starting index should be *larger* than the stopping index:

>>> list(range(2050, 2000, -4)) ?

Question: Do these function work? If so what will be the result?

>>>list(range(2000, 2050, -4))

>>> list(range(2050, 2000, 4))

Accumulating the total of range

It's possible to loop over the sequence produced by a call on range. For example,
the following program calculates the sum of the integers from 1 to 10:

```
>>> total = 0
>>> for i in range(1, 10):
... total = total + i
...
>>> total
?
```

Notice that the upper bound passed to range is 10. It's one more than the greatest integer we actually want.

Tracing the total code

To understand how the accumulating total works let us trace the code

```
total = 0
for i in range(1,10):
        total+=i
Print('Total of range = ',total)
```

Processing Lists Using Indices

In the following example we have to distinguish between the variable i, which stands for *index* and the values which refer to the items of the list.

Each index to access the items in the list:

```
>>> values = [4, 10, 3, 8, -6]
>>> for i in range(len(values)):
... print(i, values[i])
?
```

You can also use them to modify list items:

```
>>> values = [4, 10, 3, 8, -6]
>>> for i in range(len(values)):
... values[i] = values[i] * 2
...
>>> values  ?
```

Processing Parallel Lists Using Indices

Sometimes the data from one list corresponds to data from another. For example, consider these two lists:

```
>>> metals = ['Li', 'Na', 'K‘]
>>> weights = [6, 22, 39]
```

The item at index 0 of metals has its atomic weight at index 0 of weights and so on …

We would like to print each metal and its weight. To do so, we can loop over each index of the lists, accessing the items in each in parallel way:

```
>>> metals = ['Li', 'Na', 'K']
>>> weights = [6.941, 22.98976928, 39.0983]
>>> for i in range(len(metals)):
... print(metals[i], weights[i])
?
```

Nesting Loops in Loops

The block of statements inside a loop can contain another loop. In this code, the inner loop is executed once for each item of list outer:

```
>>> outer = ['Li', 'Na', 'K']
>>> inner = ['F', 'Cl', 'Br']
>>> for metal in outer:
... for halogen in inner:
... print(metal + halogen)
?
```

Generate a multiplication table via nested loops

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

Looping Over Nested Lists

- So far, we have seen Looping over lists of numbers, strings, and Booleans,

- loop could be done over lists of lists.

Example:

```
>>> elements = [['Li', 'Na', 'K'], ['F', 'Cl', 'Br']]
>>> for inner_list in elements:
... print(inner_list)
?
```

This works exactly like nested loop:

```
>>> elements = [['Li', 'Na', 'K'], ['F', 'Cl', 'Br']]
>>> for inner_list in elements:
... for item in inner_list:
... print(item)
```

Looping Over Ragged Lists

- R*agged lists* are nested lists with inner lists of varying lengths.
- Data isn't uniform;
- Not like parallel where you can assemble matching fields,
- trying to assemble a list of email addresses for data where some addresses are missing

Ex:

```
>>> info = [['Isaac Newton', 1643, 1727],
... ['Charles Darwin', 1809, 1882],
... ['Alan Turing', 1912, 1954, 'alan@bletchley.uk']]
>>> for item in info:
... print(len(item))
```

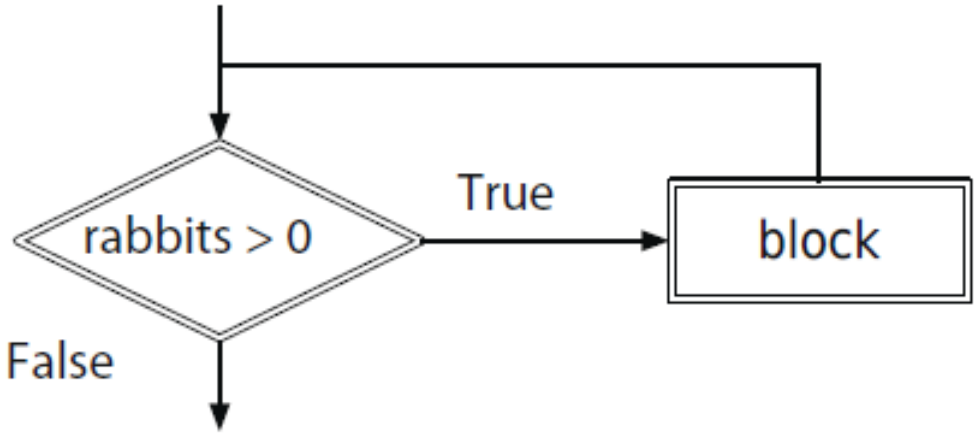
Looping Until a Condition Is Reached

- for loops are useful only if you know how many iterations of the loop you need.
- In some situations, it is not known in advance how many loop iterations to execute.
- In a game program, for example, you can't know whether a player is going to want to play again or quit.
- In these situations, we use a while loop.

The general form of a while loop is as follows:

while *«expression»*:
«block»

Looping Until a Condition Is Reached



Ex:

time = 0

population = 1000 *# 1000 bacteria to start with*

growth_rate = 0.21 *# 21% growth per minute*

Infinite Loops

Exercise: let us do this code and learn from it what is the infinite loop

Find the growth of bacteria (population) till day 5

Do it?

Repetition Based on User Input

EX: We will ask the user to enter a chemical formula, and our program, which is saved in a file named formulas.py, will print its name. This should continue until the user types quit:

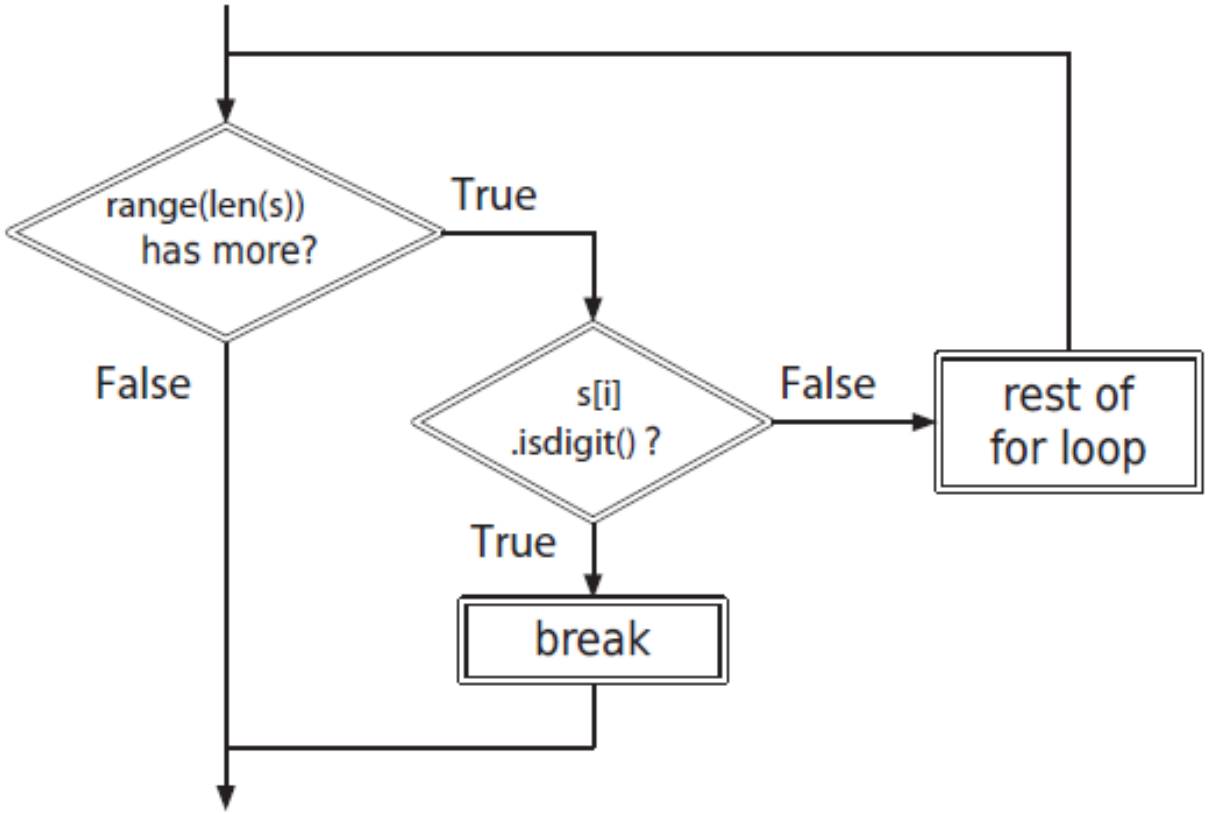
```
text = ""
while text != "quit":
    text = input("Please enter a chemical formula (or 'quit' to exit): ")
    if text == "quit":
        print("…exiting program")
    elif text == "H2O":
        print("Water")
    elif text == "NH3":
        print("Ammonia")
    elif text == "CH4":
        print("Methane")
    else:
        print("Unknown compound")
```

Controlling Loops Using Break and Continue

Loops keep executing all the statements in their body on each iteration. However, sometimes it is handy to be able to break that rule. Python provides two ways of controlling the iteration of a loop: break, which terminates execution of the loop immediately, and continue, which skips ahead to the next iteration.

Controlling Loops Using Break and Continue

Notice that because the loop terminates early, we were able to simplify the if statement condition. As soon as digit_index is assigned to digit , the loop terminates, so it isn't necessary to check the rest of the string.

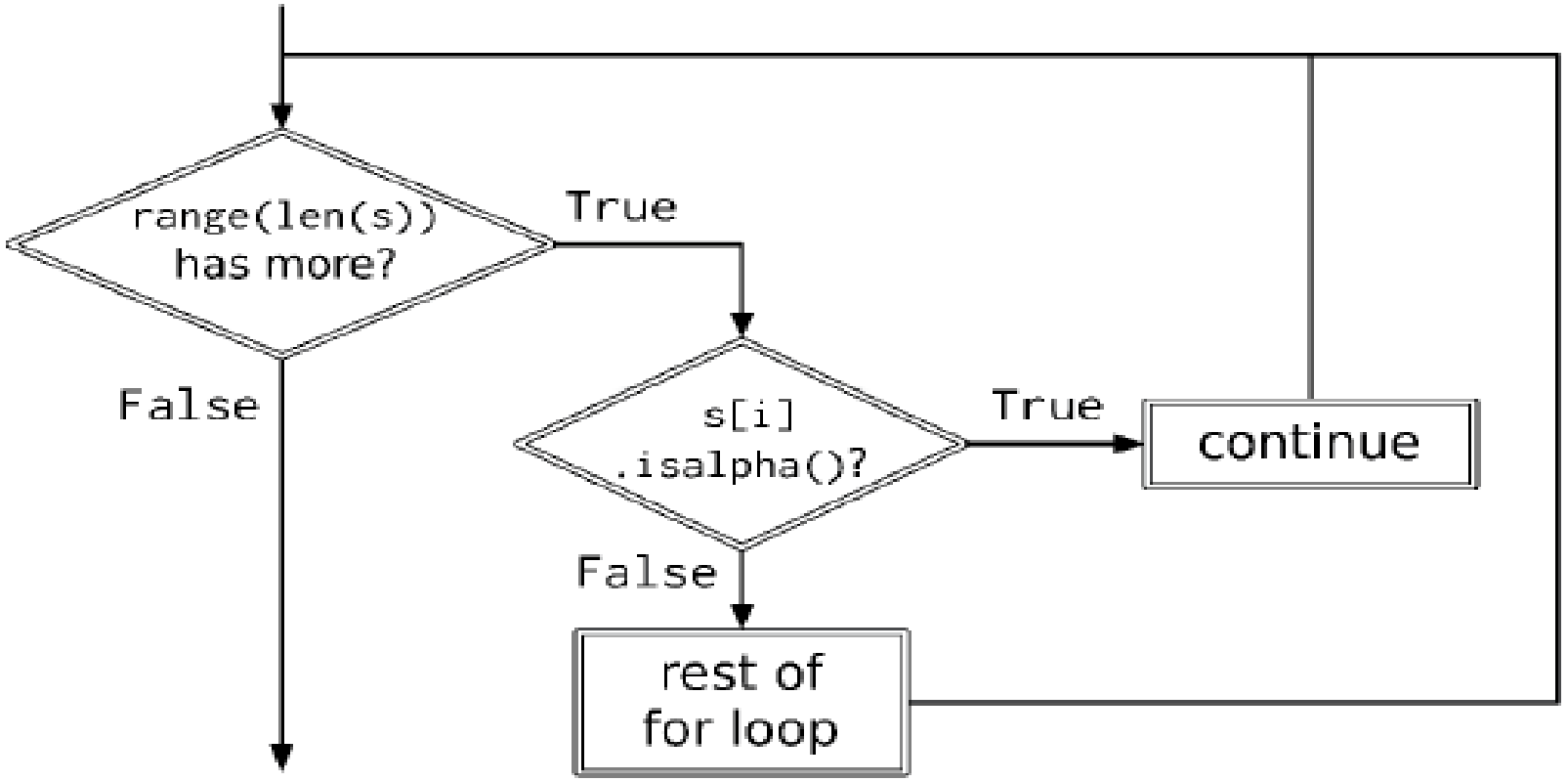


The Continue Statement

Another way to bend the rules for iteration is to use the continue statement, which causes Python to skip immediately ahead to the next iteration of a loop.

The Continue Statement

When continue is executed, it *immediately* begins the next iteration of the loop. All statements in the loop body that appear after it are skipped,



The return statement in a loop

Duplicate items

Let us solve this problem for a given list
return true if there is at least two elements in the list are equal , otherwise return false

Duplicate items in non sorted and sorted list

Let us see the difference from running the following two codes
First let us recall the duplicate code without sort

In class exercise

Give a list of random numbers
Embed negative number in the list any where ex:

List = [28, 33,22,55,33,66,-77,33,55,33]

Write Python code to find the sum of even numbers
Quit the loop when you encountered negative value in the list