Chapter 7
Iteration and Search

A. Downey, *Think Python: How to Think Like a Computer Science,* 3rd ed., O'Reilly, 2024.

https://allendowney.github.io/ThinkPython/

Contents

1. Loops and Strings

- **for** loop:

```
for i in range(3):
    print(i, end=' ')
```

```
0 1 2
```

```
for letter in 'Gadsby':
    print(letter, end=' ')
```

```
G a d s b y
```

  - This version uses the keyword argument **end**, so the print function puts a space after each number rather than a newline.
  - The variable defined in a for loop is called the **loop variable**.

1. Loops and Strings

```
def has_e(word):
    for letter in word:
        if letter == 'E' or letter == 'e':
            return True
    return False
```

```
has_e('Gadsby')
```

```
False
```

```
has_e('Emma')
```

```
True
```

1. Loops and Strings

- **while** loop
  - A **while** loop is used to execute a block of statements repeatedly while a given condition is satisfied. When the condition becomes false, the **while** loop stops and the line immediately after the loop in the program is executed.

```
1 count = 0
2 while (count < 5):
3     count = count + 1
4     print("Hello World!")
```

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

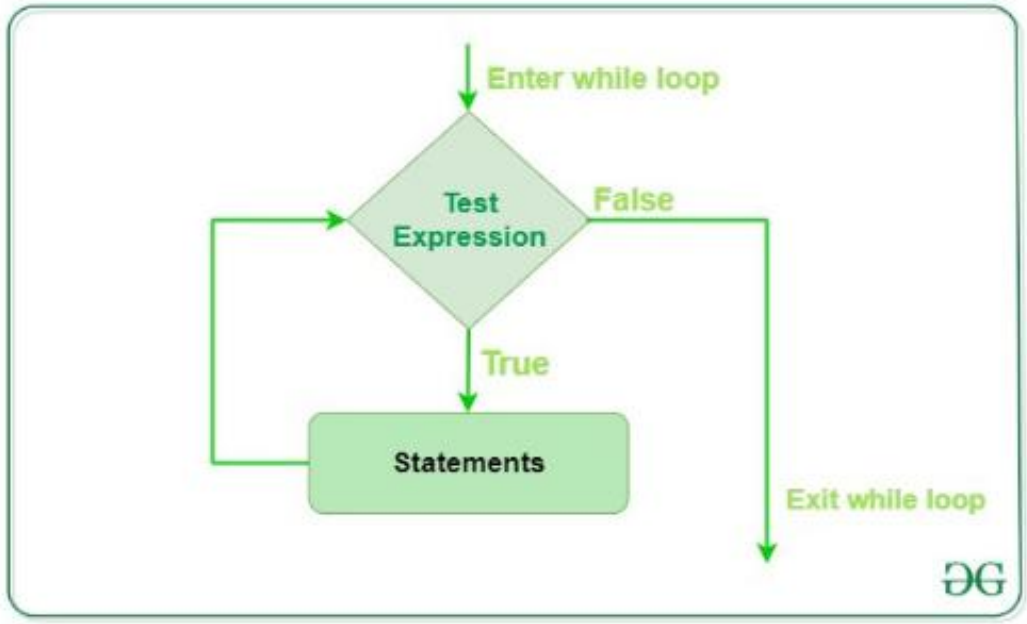
  - The condition (count < 5) is checked. If the condition is satisfied, the body of the while loop is executed and then the condition is cheched again. If it is satisfied, the body is executed again ...

1. Loops and Strings

- **while** loop

```
while expression:
    statement(s)
```

Flowchart of Python While Loop



1. Loops and Strings

- Infinite **while** loop

```
1 count = 0
2 while (count == 0):
3     print("Hello World!")
```

```
Hello World!
Hello World!
Hello World!
```

1. Loops and Strings

- Loop control statements
  - **continue** statement: The **continue** statement skips the current iteration and returns the control to the beginning of the loop for the next iteration.

```
1 for letter in 'Hello':
2     if letter == 'e' or letter == 'E':
3         continue
4     print('Current Letter :', letter)
5
```

```
Current Letter : H
Current Letter : l
Current Letter : l
Current Letter : o
```

1. Loops and Strings

- **break** Statement: The **break** statement in Python brings control out of the loop (break the loop).

```
1 for letter in 'Hello':
2     if letter == 'e' or letter == 'E':
3         break
4     print('Current Letter :', letter)
5
```

```
Current Letter : H
```

1. Loops and Strings

- **break** Statement

2. Reading the Word List

- Reading a file called *words.txt*
  - 1. Open the file:
    - Use the built-in function **open**, which takes the name of the file as a parameter and returns a **file object** we can use to read the file.
  - 2. Reading the file:
    - The file object provides a function called **readline**, which reads characters from the file until it gets to a newline and returns the result as a string.
      - **readline** is a method associated with an object **file_object** so we call it using the name of the object, the dot operator, and the name of the method.

```
1 file_object = open('words.txt')
2
3 file_object.readline()
```

```
'aa\n'
```

words.txt ×

```
aa
aah
aahed
aahing
aahs
aal
aalii
aaliis
aals
aardvark
aardvarks
aardwolf
aardwolves
aas
aasvogel
aasvogels
aba
abaca
abacas
abaci
aback
abacus
abacuses
abaft
abaka
abakas
abalone
abalones
abamp
abampere
abamperes
```

2. Reading the Word List

- The first word in the list is **“aa”** which is a type of lava. The sequence \n represents the newline character that separates this word from the next.
- The file object keeps track of where it is in the file, so if you call readline again, you get the next word:

```
1 line = file_object.readline()
2 line
```

```
'aah\n'
```

- To remove the newline from the end of the word, we can use **strip**, which is a method associated with strings, so we can call it like this:
  - **strip** removes whitespace characters—including spaces, tabs, and newlines—from the beginning and end of the string.

```
1 word = line.strip()
2 word
```

```
'aah'
```

words.txt ×

```
aa
aah
aahed
aahing
aahs
aal
aalii
aaliis
aals
aardvark
aardvarks
aardwolf
aardwolves
aas
aasvogel
aasvogels
aba
abaca
abacas
abaci
aback
abacus
abacuses
abaft
abaka
abakas
abalone
abalones
abamp
abampere
abamperes
```

2. Reading the Word List

- You can also use a file object as part of a **for** loop. This program reads *words.txt* and prints each word, one per line:

```
1 for line in open('words.txt'):
2     word = line.strip()
3     print(word)
```

```
aa
aah
aahed
aahing
aahs
aal
aalii
aaliis
aals
aardvark
aardvarks
aardwolf
aardwolves
aas
aasvogel
```

words.txt ×

```
aa
aah
aahed
aahing
aahs
aal
aalii
aaliis
aals
aardvark
aardvarks
aardwolf
aardwolves
aas
aasvogel
aasvogels
aba
abaca
abacas
abaci
aback
abacus
abacuses
abaft
abaka
abakas
abalone
abalones
abamp
abampere
abamperes
```

3. Updating Variables

- For example, here is an initial assignment that creates a variable:

```
1 x = 5
2 x
```

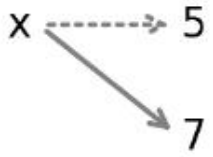
```
5
```

- And here is an assignment that changes the value of a variable:

```
1 x = 7
2 x
```

```
7
```

- The following figure shows what these assignments look like in a state diagram:



The dotted arrow indicates that **x** no longer refers to 5. The solid arrow indicates that it now refers to 7.

3. Updating Variables

- A common kind of assignment is an **update:**

```
1 x = x + 1
2 x
```

```
8
```

- Increasing the value of a variable is called an **increment**; decreasing the value is called a **decrement**.

```
1 y = 0
2 y = y + 1
3 y
```

```
1
```

```
1 y = y - 1
2 y
```

```
0
```

4. Looping and Counting

- The following program counts the number of words in the word list:

```
1 total = 0
2
3 for line in open('words.txt'):
4     word = line.strip()
5     total = total + 1
```

```
1 total
```

```
113783
```

  - A variable like **total**, used to count the number of times something happens, is called a **counter**.

4. Looping and Counting

- We can add a second counter to the program to keep track of the number of words that contain an “e”:

```
def has_e(word):
    for letter in word:
        if letter == 'E' or letter == 'e':
            return True
    return False
```

```
total = 0
count_e = 0

for line in open('words.txt'):
    word = line.strip()
    total = total + 1
    if has_e(word):
        count_e = count_e + 1
```

```
count_e
```

```
76162
```

5. The **in** Operator

- Python provides an operator, **in**, that checks whether a character appears in a string:

```
1 word = 'Gadsby'
2 'e' in word
```

```
False
```

- So we can rewrite **has_e** like this:

```
1 def has_e(word):
2     if 'E' in word or 'e' in word:
3         return True
4     else:
5         return False
```

```
1 def has_e(word):
2     for letter in word:
3         if letter == 'E' or letter == 'e':
4             return True
5     return False
```

```
1 def has_e(word):
2     return 'E' in word or 'e' in word
```

5. The **in** Operator

- Method **lower** of a string object:
  - Returns a new string that converts the letters in a string to lowercase

```
1 word = 'Gadsby'
2 word.lower()
```

```
'gadsby'
```

    - **lower** makes a new string—it does not modify the existing string—so the value of **word** is unchanged ‘Gadsby’.

- Here’s how we can use lower in **has_e**:

```
1 def has_e(word):
2     return 'e' in word.lower()
```

6. Search

- Let's write a function called **uses_any** that takes two string parameters called **word** and **letters**. It returns **True** if the **word** uses any of the **letters**, and **False** otherwise:

```
1 def uses_any(word, letters):
2     for letter in word.lower():
3         if letter in letters.lower():
4             return True
5     return False
```

```
1 uses_any('banana', 'aeiou')
```

```
True
```

```
1 uses_any('apple', 'xyz')
```

```
False
```

- **uses_any** converts word and letters to lowercase, so it works with any combination of cases:

```
1 uses_any('Banana', 'AEIOU')
```

```
True
```

6. Search

- It loops through the letters in word and checks them one at a time. If it finds one that appears in letters, it returns `True` immediately. If it gets all the way through the loop without finding any, it returns **`False`**.
- This pattern is called a **linear search**.

7. Doctest

- To run these tests, we have to import the **doctest** module and run a function called **run_docstring_examples**.

```
1 from doctest import run_docstring_examples
2
3 run_docstring_examples(uses_any, globals())
```

- **un_docstring_examples** finds the expressions in the docstring and evaluates them.
  - If the result is the expected value, the test passes. Otherwise it fails.
  - If all tests pass, run_doctests displays no output—in that case, no news is good news.

7. Doctest

- It is also possible to use a docstring to *test* a function. Here's a version of **uses_any** with a docstring that includes tests:

```
1  def uses_any(word, letters):
2      """Checks if a word uses any of a list of letters.
3      >>> uses_any('banana', 'aeiou')
4      True
5      >>> uses_any('apple', 'xyz')
6      False
7      """
8      for letter in word.lower():
9          if letter in letters.lower():
10             return True
11     return False
```

  - Each test begins with **>>>**, which is used as a prompt in some Python environments to indicate where the user can type code.
  - In a doctest, the prompt is followed by an expression, usually a function call. The following line indicates the value the expression should have if the function works correctly.