OReilly - 2024-02February-15

February 22, 2024

1 Agenda: Day 2

1. Q&A from last time
2. Loops
   - `for` on strings (and other iterables)
   - `for` on numbers (with `range`)
   - The index (or lack thereof)
   - `while` loops
   - Controlling our loops with `break` and `continue`
3. Lists
   - What are lists?
   - Compare lists with strings
   - Lists are mutable – and what this means
4. Turning strings into lists, and back
   - `str.split`
   - `str.join`
5. Tuples
   - What are tuples, and how do they fit into what we've already learned?
   - Unpacking

2 Loops

One of the most important ideas in the world of programming is the "DRY rule," meaning, "Don't repeat yourself."

The idea is that if you have the same code running several times in a row, then you shouldn't be doing that. You should find a way to contract that code into a smaller, more elegant way of expressing it.

Why?

- It'll take less time to write
- It'll take less time to read/understand/maintain
- You avoid the potential issue of having multiple, different versions of the same code

[1]:
```
# Example: Let's print every character in a string

s = 'abcd'
```

```
print(s[0])
print(s[1])
print(s[2])
print(s[3])
```

```
a
b
c
d
```

[2]:
```
# there is a better way -- a loop!
# a loop allows us to repeat (more or less) the same command on several␣
 ↪different objects
# in this case, we want to repeat the same call to "print" on each character

# Python has two kinds of loops: for and while

for one_character in s:
    print(one_character)
```

```
a
b
c
d
```

3 Syntax of the `for` loop

- We need to write the word `for`
- We need to indicate into what variable will each iteration (time running the loop) be assigned? This is known as the "loop variable."
- The word `in`
- The object on which we're iterating
- A colon at the end of the line
- The next line(s) must be indented – that's the loop body
- The loop body can be as long as we want; it will run once for each iteration of the loop

The loop body can contain *any* code we want: - Assignment - `print` - `input` - `if/else` - Another `for` loop ("nested loop")

4 What's really happening in our `for` loop?

1. `for` asks the object at the end of the line: Are you iterable? Do you know how to behave in a `for` loop?
   - If the answer is "no," the loop exits right away
2. If the object is iterable, then `for` says: Give me the next value. What comes next?
   - If there are no more values, then the object says so, and the `for` loop ends.
3. The object (`s`, in this case) gives `for` its next value, which is assigned to our variable (in this case, `one_character`)

4. Once that assignment has taken place, the loop body is executed with the variable assigned.
5. When the loop body is done, we go back to step 2.

MANY MANY MANY people in my courses believe that we get one character at a time from `s`, because we named our variable `one_character`. This is *NOT TRUE*. We will get one character at a time from a string, no matter what we call the variable. The variable name is 100% for us to understand the program; Python couldn't care less what we call it.

`one_character` is a variable ("loop variable" or "iteration variable"), and it's assigned a value once per iteration, by the `for` loop. We aren't assigning it a value directly.

[3]:
```
for one_character in s:
    print(one_character)
```

```
a
b
c
d
```

[4]:
```
for one_terabyte in s:
    print(one_terabyte)
```

```
a
b
c
d
```

[5]:
```
for one_pizza in s:
    print(one_pizza)
```

```
a
b
c
d
```

[6]:
```
print(s)
```

```
abcd
```

5 Exercise: Vowels, digits, and others

1. Define three variables, `vowels`, `digits`, and `others`. Assign 0 to each of them.
2. Ask the user to enter a string.
3. Go through the string, one character at a time:
   - If the character is a vowel, add 1 to `vowels`.
   - If the character is a digit, add 1 to `digits`.
   - Otherwise, add 1 to `others`.
4. Print the values of each of these three variables.

Example:

```
Enter some text: hello!! 123
vowels: 2
digits: 3
others: 6
```

Hints/reminders: - You can get input from the user with `input` - You can check character membership in a string with `in` - You can check if a string contains only digits 0-9 with the `str.isdigit` method

[7]:
```
vowels = 0
digits = 0
others = 0

s = input('Enter some text: ').strip()

for one_character in s:
    if one_character in 'aeiou':
        vowels += 1    # add 1 to the vowels count
    elif one_character.isdigit():
        digits += 1    # add 1 to the digits count
    else:
        others += 1

print(f'vowels = {vowels}')
print(f'digits = {digits}')
print(f'others = {others}')
```

```
Enter some text:  hello!! 123

vowels = 2
digits = 3
others = 6
```

[8]:
```
# what if I want to repeat something a number of times?

print('Hooray!')
print('Hooray!')
print('Hooray!')
```

```
Hooray!
Hooray!
Hooray!
```

[9]:
```
# no no no let's "DRY up" this code

for counter in 3:      # for loop asked 3, an integer: Are you iterable?
    print('Hooray!')  # sadly, the answer is "no"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
Cell In[9], line 3
      1 # no no no let's "DRY up" this code
----> 3 for counter in 3:
      4     print('Hooray!')

TypeError: 'int' object is not iterable
```

[10]:
```
# we can iterate over a "range" object, which we can create by calling range(3)

for counter in range(3):  # range(3) is iterable!
    print('Hooray!')
```

```
Hooray!
Hooray!
Hooray!
```

[11]:
```
# what are we getting with each iteration here?

for counter in range(3):           # here, I chose to use the variable name␣
 ↪"counter"
    print(f'{counter} Hooray!')    #   I could have said "i" or "j" or "index"␣
 ↪or "elephant"
```

```
0 Hooray!
1 Hooray!
2 Hooray!
```

6 Using `range`

When we iterate over `range`, we get the number of iterations we asked for in the argument.

We start counting with 0, though. So if we say `range(5)`, we'll get 5 iterations, and the number we get will be 0, 1, 2, 3, and 4. The maximum number will always be 1 less than what we gave to `range` as an argument.

7 Exercise: Sum numbers

1. Define a variable, `total`, and set it to 0.
2. Ask the user, how many numbers they're going to enter. Assign that to `count`.
3. Using `for` and `range`, ask the user to enter `count` integers.
   - Get their input with `input`
   - Turn it into an integer with `int`
   - (If you want, check that it contains only digits, and can be turned into an `int`, with `isdigit`
4. Print the total.

Example:

```
How many numbers? 4
Enter number 0: 10
Enter number 1: 20
Enter number 2: 30
Enter number 3: hello
    hello is not a number
Total is 60
```

[12]:
```
s = '1'   # this is a string containing a single character, the digit 1  --it's not an integer!

s.isdigit()
```

[12]:
```
True
```

[13]:
```
s += 1   # this means: add the integer 1 to the string '1' ... which won't work!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[13], line 1
----> 1 s += 1   # this means: add the integer 1 to the string '1' … which won't work!

TypeError: can only concatenate str (not "int") to str
```

[14]:
```
# if a string returns True for isdigit, then we know we can turn it into an integer with int()

int(s) + 1    # now we're adding an integer to an integer...
```

[14]:
```
2
```

[18]:
```
total = 0

s = input('How many numbers? ').strip()  # this is a string, because input returns strings!
count = int(s)    # get an integer based on s, and assign to count

for index in range(count):
    user_input = input(f'[{index}] Enter number: ').strip()     # get the number from the user

    if user_input.isdigit():
        total += int(user_input)                                # turn it into an int, and add to total
    else:
```

```
        print(f'\t{user_input} is not numeric; ignoring')

print(f'total = {total}')
```

```
How many numbers?  4
[0] Enter number:  2
[1] Enter number:  4
[2] Enter number:  hello

        hello is not numeric; ignoring

[3] Enter number:  6

total = 12
```

[20]:
```
# what happens if the user gives a non-integer to the inputs in the above␣
 ↪exercise?

total = 0

s = input('How many numbers? ').strip()  # this is a string, because input␣
 ↪returns strings!
if s.isdigit():
    count = int(s)    # get an integer based on s, and assign to count

    for index in range(count):
        user_input = input(f'[{index}] Enter number: ').strip()     # get the␣
 ↪number from the user

        if user_input.isdigit():
            total += int(user_input)                                    # turn␣
 ↪it into an int, and add to total
        else:
            print(f'\t{user_input} is not numeric; ignoring')

    print(f'total = {total}')
else:
    print(f'{s} is not numeric; ending now!')
```

```
How many numbers?  2
[0] Enter number:  5
[1] Enter number:  hello

        hello is not numeric; ignoring
total = 5
```

[ ]:
```

```

8 Next up

1. Index (or lack thereof)
2. Loop controls

If you come from another programming language (e.g., C or Java), then you're used to `for` loops that look a bit different: In those languages, you start your variable with 0, and then slowly but surely increment to a maximum number. You use that variable to retrieve values from a string, array, etc.

The fact that Python doesn't have an index when we run a `for` loop seems really strange to such people.

Think of it this way:

- In C, we need to use the index in order to retrieve from a string. The string isn't smart enough to give us its characters, one at a time. So to make up for it, our loop needs to be smarter.
- In Python, the string is pretty smart, and knows how to behave. So our loop can be simpler/dumber, doing very little other than just asking the string to give us its successive characters.

But… there are times when we might want an index, if only to display characters with their indexes.

[22]:
```
# what if we want to print each character with its index in the string?
# option 1: do it manually

s = 'abcd'
index = 0     # here, we're just setting a variable

for one_character in s:
    print(f'{index}: {one_character}')

    # each iteration is from the next character
    index += 1     # this adds 1 to index
```

```
0: a
1: b
2: c
3: d
```

[26]:
```
# option 2: Use enumerate
# the syntax looks weird -- we'll get to it later today

# the enumerate function is something you run on an iterable value
# instead of giving one thing with each iteration (i.e., the current character),
# enumerate returns *two* things -- the current index (which it calculates) and
# the current character

# enumerate returns two values: First the index, then the original one
# this is considered "Pythonic" -- meaning, idiomatic in the Python community
```

```
s = 'abcd'
for index, one_character in enumerate(s):
    print(f'{index}: {one_character}')
```

```
0: a
1: b
2: c
3: d
```

[25]:
```
# this is a total, crazy exaggeration ... and yet, it works

s = 'abcd'
for book, tissue in enumerate(s):
    print(f'{book}: {tissue}')
```

```
0: a
1: b
2: c
3: d
```

9 Exercise: Powers of 10

As you might know, numbers in the decimal system are built out of powers of 10. If I have the number 1234, we could restate that as:

```
(1 * 10**3) + (2 * 10**2) + (3 * 10**1) + (4 * 10**0)
```

1. Ask the user to enter a number.
2. Print the number in the above expanded format, showing the digit, times the power of 10 we want
3. (It's OK to print the output on separate lines.)

Remember: - You can get the length of a string with `len` - Don't forget to convert strings into integers if you want to calculate with them – but if you're just showing the characters, then you don't really need to make that conversion.

[30]:
```
s = input('Enter a number: ').strip()

for index, one_digit in enumerate(s):
    print(f'{one_digit} * 10**{len(s) - index - 1}')
```

```
Enter a number:  1234

1 * 10**3
2 * 10**2
3 * 10**1
4 * 10**0
```

[34]:
```
# AB

number = '1234'

for index, chiffre in enumerate(number):
    x = len(number) - index - 1
    print(f"{chiffre} * 10** {x}")
    index+=1
```

```
1 * 10** 3
2 * 10** 2
3 * 10** 1
4 * 10** 0
```

[37]:
```
# print everything on one line
# cheap method (better method, with join, will come later)

output = ''
s = input('Enter a number: ').strip()

for index, one_digit in enumerate(s):
    output += f'({one_digit} * 10**{len(s) - index - 1}) '

print(output)
```

```
Enter a number:  1234
(1 * 10**3) (2 * 10**2) (3 * 10**1) (4 * 10**0)
```

10 Why is `len` a function, and not a method?

People always think that given a string `s`, they can get the length with `s.len()`. But this *DOES NOT WORK*, because there is no such method. Rather, `len` is a function, which you call as `len(s)`.

Why? There's no truly great reason.

11 Controlling our loops

Normally, a `for` loop runs until the end. What if you want to stop it prematurely?

- Sometimes, you want to stop the entire loop – exit the loop, right away. This can be done with `break`.
- Sometimes, you want to stop the current iteration, and move onto the next one. This can be done with `continue`.

[39]:
```
# continue is often used to identify a value that we don't want/need/care␣
 ↪about, and
# to move onto the next iteration of our for loop
```

```
s = 'abcd'
look_for = 'c'

for one_character in s:
    if one_character == look_for:
        continue     # this means: finish the current iteration, and go onto␣
 ↪the next one

    print(one_character)
```

```
a
b
d
```

[41]:
```
# by contrast, break means: We have reached the end, and there's no need to␣
 ↪continue
# iterating past this point.

s = 'abcd'
look_for = 'c'

for one_character in s:
    if one_character == look_for:
        break

    print(one_character)

print('Done.')
```

```
a
b
Done.
```

12 `while` loops

`for` loops execute once for each element in the object over which we're iterating. They're perfect when we know how many iterations we want.

But what if we don't know how many iterations we want? What if we know the situation when we'll want to stop, but we can't articulate when that'll be?

- We're waiting for a particular type of input from the user
- We're waiting to reach a certain goal

A `while` loop is like an `if` statement – it has a condition, and if the condition is `True`, it executes the block. The difference is that an `if` statement runs once. A `while` loop will keep running so long as its condition is `True`.

This means that if you say

`while True:`

You now have an infinite loop! Make sure that you have a `break` in your loop body that can be reached so that you don't end up with such a loop.

[42]:
```
x = 5

print('Before')
while x > 0:
    print(x)
    x -= 1    # this means: x = x - 1
print('After')
```

```
Before
5
4
3
2
1
After
```

[43]:
```
# we can combine while loops and break

while True:   # yes, I'm courting danger!
    name = input('Enter your name: ').strip()

    if name == '':    # empty string?
        break         # exit!

    print(f'Hello, {name}!')
```

```
Enter your name:  Reuven
Hello, Reuven!
Enter your name:  world
Hello, world!
Enter your name:  out there
Hello, out there!
Enter your name:
```

13 Exercise: Sum to 100

1. Set `total` to be 0.
2. Ask the user, repeatedly, to enter a number.
   - If it's not digits, then scold the user.
3. Turn the user's input into an `int` and add to `total`.

4. When `total` is 100 or greater, exit from the loop. This can be the condition in your `while`, or it can be inside of the loop body, with `break`.

Example:

```
Enter a number: 50
total is 50
Enter a number: 20
total is 70
Enter a number: 40
total is 110
```

[44]:
```
# my preferred way

total = 0

while total < 100:
    s = input('Enter a number: ').strip()

    if s.isdigit():
        total += int(s)
        print(f'total is {total}')
    else:
        print(f'{s} is not numeric')
```

```
Enter a number:  20
```

```
total is 20
```

```
Enter a number:  30
```

```
total is 50
```

```
Enter a number:  50
```

```
total is 100
```

[ ]:
```
# my preferred way

total = 0

while True:
    s = input('Enter a number: ').strip()

    if s.isdigit():
        total += int(s)
        print(f'total is {total}')

        if total >= 100:
            break
    else:
```

```
        print(f'{s} is not numeric')
```

[45]:
```
# MG
total = 0
for i in range(100):
 inp = input('enter a number please')
 for j in inp:
  if j in 0-9:
    break

print(f'thanks')
```

```
enter a number please 2
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[45], line 5
      3 for i in range(100):
      4  inp = input('enter a number please')
----> 5  if j in inp:
      6   if j in 0-9:
      7     break

NameError: name 'j' is not defined
```

14 Next up

1. Lists
   - Creating
   - Retrieving
   - Iterating
   - Mutable data
2. Turning lists to strings with `str.join`
3. Turning strings into lists with `str.split`
4. Tuples

15 Lists

Strings are "containers." They contain individual characters. Sometimes, we want a more generic container that can contain anything.

That's what lists are for. They can contain absolutely any Python values we want. They're very similar to what other languges would call "arrays," but we use the term "list" because technically speaking, arrays are different.

To create a list, we use `[]`, with commas between the elements.

In theory, you can put any values you want in a list. Traditionally, though, we only create lists where all values are of the same type: List of strings, list of integers, list of floats, list of lists.

[46]:
```
mylist = [10, 20, 30, 40, 50]

type(mylist)   # what kind of value is this?
```

[46]:
```
list
```

[47]:
```
mylist = []   # this is the empty list
len(mylist)
```

[47]:
```
0
```

[48]:
```
mylist = [10, 20, 30, 40, 50]
len(mylist)
```

[48]:
```
5
```

[49]:
```
mylist[0]
```

[49]:
```
10
```

[50]:
```
mylist[1]
```

[50]:
```
20
```

[51]:
```
mylist[-1]
```

[51]:
```
50
```

[53]:
```
# get a slice
mylist[1:4]
```

[53]:
```
[20, 30, 40]
```

[54]:
```
# search in a list
30 in mylist
```

[54]:
```
True
```

[55]:
```
100 in mylist
```

[55]:
```
False
```

[56]:
```
# iterate over a list
```

```
for one_item in mylist:
    print(one_item)
```

```
10
20
30
40
50
```

[57]:
```
mylist = [10, 20, 30]
biglist = [mylist, mylist, mylist]

len(biglist)   # how many elements?
```

[57]:
```
3
```

[58]:
```
biglist
```

[58]:
```
[[10, 20, 30], [10, 20, 30], [10, 20, 30]]
```

[59]:
```
# you might remember that we cannot change strings, because they are immutable
s = 'abcd'
s[0] = '!'    # we cannot do this!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[59], line 3
      1 # you might remember that we cannot change strings, because they are␣
 ↪immutable
      2 s = 'abcd'
----> 3 s[0] = '!'    # we cannot do this!

TypeError: 'str' object does not support item assignment
```

[60]:
```
# but... lists are mutable! We can change them!

mylist[0] = '!'
mylist
```

[60]:
```
['!', 20, 30]
```

[61]:
```
biglist
```

[61]:
```
[['!', 20, 30], ['!', 20, 30], ['!', 20, 30]]
```

16 How can we change lists?

1. We can assign to a particular element, using the index

   mylist[0] = 'hello'

2. We can add one element to the end of the list, using `list.append`

   mylist = [10, 20, 30] mylist.append(40) print(mylist) # [10, 20, 30, 40]

3. We can add multiple values to the end of a list with `+=`

   mylist = [10, 20, 30] mylist += [40, 50, 60] # += will iterate over the thing to its right, appending each to mylist print(mylist) # [10, 20, 30, 40, 50, 60]

4. We can remove the final element with `list.pop`:

   mylist = [10, 20, 30] mylist.pop() print(mylist) # [10, 20]

[62]:
```
mylist = [10, 20, 30]
mylist.append(40)
mylist
```

[62]:
```
[10, 20, 30, 40]
```

[63]:
```
mylist.append('abcd')
mylist
```

[63]:
```
[10, 20, 30, 40, 'abcd']
```

[64]:
```
mylist += [50, 60, 70]   # we'll run a for loop on [50, 60, 70], appending each␣
 ↪element to mylist
mylist
```

[64]:
```
[10, 20, 30, 40, 'abcd', 50, 60, 70]
```

[65]:
```
# what if I try to use += with a number?

mylist += 80   # this won't work! Because you cannot run a for loop on 80
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[65], line 3
      1 # what if I try to use += with a number?
----> 3 mylist += 80   # this won't work! Because you cannot run a for loop on 80

TypeError: 'int' object is not iterable
```

[66]:
```
mylist += 'abcd'   # what happens here?
mylist
```

[66]:
```
[10, 20, 30, 40, 'abcd', 50, 60, 70, 'a', 'b', 'c', 'd']
```

[67]:
```
mylist.pop()
```

[67]:
```
'd'
```

[68]:
```
mylist
```

[68]:
```
[10, 20, 30, 40, 'abcd', 50, 60, 70, 'a', 'b', 'c']
```

It's pretty common to start with an empty list at the top of a program, and then go through inputs of various sorts (from the user or a file) and then categorizing the values we got, and putting them on different lists.

17 Exercise: Vowels, digits, and others – list edition

1. Define three empty lists, `vowels`, `digits`, and `others`.
2. Ask the user to enter a string.
3. Go through the string, one character at a time.
   - If the character is a vowel, append to the end of `vowels`.
   - If the character is a digit, append to the end of `digits`.
   - Otherwise, append to the end of `others`.
4. Print all three lists.

[69]:
```
vowels = []
digits = []
others = []

s = input('Enter text: ').strip()

for one_character in s:
    if one_character in 'aeiou':
        vowels.append(one_character)
    elif one_character.isdigit():
        digits.append(one_character)
    else:
        others.append(one_character)

print(f'vowels = {vowels}')
print(f'digits = {digits}')
print(f'others = {others}')
```

```
Enter text:  hello!! 123
```

```
vowels = ['e', 'o']
digits = ['1', '2', '3']
others = ['h', 'l', 'l', '!', '!', ' ']
```

[70]:
```
# given an integer, I can get a string or a float from it

n = 10
str(n)   # give me a string, based on n
```

[70]: '10'

[71]:
```
float(n)  # give  me a float, based on n
```

[71]: 10.0

[72]:
```
# given a string, I can get an int or float from it
s = '1234'

int(s)
```

[72]: 1234

[73]:
```
float(s)
```

[73]: 1234.0

[75]:
```
# the general rule is: If I have data in a variable x, and I want it to be in a␣
 ↪different type T,
# I can run T(x) and get a new value back, based on x, of type T

# what if I have a string, and I want to turn it into a list
s = 'abcd'
list(s)    # list runs a for loop
```

[75]: ['a', 'b', 'c', 'd']

[76]:
```
# what if my string looks a bit different?

s = 'abcd:ef:ghi'

list(s)
```

[76]: ['a', 'b', 'c', 'd', ':', 'e', 'f', ':', 'g', 'h', 'i']

[77]:
```
# let's break it apart on :

s.split(':')   # the split method runs on a string, and returns a new list of␣
 ↪strings, separated by the delimiter
```

[77]: ['abcd', 'ef', 'ghi']

[78]:
```
s.split('d')  # weird, but it works
```

[78]:
```
['abc', ':ef:ghi']
```

[79]:
```
# what if I get a bunch of words from the user?

s = 'this is a bunch of words in my course'
s.split(' ')  # use space as a delimiter, get back a list of strings (words)
```

[79]:
```
['this', 'is', 'a', 'bunch', 'of', 'words', 'in', 'my', 'course']
```

[80]:
```
# what if there are different numbers of spaces between words?
s = 'this  is a   bunch of words    in my   course'

s.split(' ')   # we get a list of strings back, based on cutting every time we␣
 ↪see ' '
```

[80]:
```
['this',
 '',
 'is',
 'a',
 '',
 '',
 'bunch',
 'of',
 'words',
 '',
 '',
 '',
 'in',
 'my',
 '',
 '',
 'course']
```

[81]:
```
# Python has a solution to this: If you invoke str.split with *no* argument,
# then it'll take any whitespace, of any length, and use it as one delimiter

s.split()  # look, no argument!
```

[81]:
```
['this', 'is', 'a', 'bunch', 'of', 'words', 'in', 'my', 'course']
```

[82]:
```
# you might remember, from last week, we wrote a Pig Latin translator

word = input('Enter a word: ').strip()
if word[0] in 'aeiou':
    print(word + 'way')    # starts with a vowel? Add 'way'
else:
    print(word[1:] + word[0] + 'ay')  # otherwise, move first letter to the␣
 ↪end, and add 'ay'
```

```
Enter a word:  right
```

```
ightray
```

18 Exercise: Pig Latin sentence!

Change the code from last week, such that the user can enter a *sentence* in English (all lowercase, no punctuation), and we print the translation into Pig Latin for each word.

It's totally fine for each output word to be on a line by itself.

[83]:
```
sentence = input('Enter a sentence: ').strip()

for word in sentence.split():
    if word[0] in 'aeiou':
        print(word + 'way')
    else:
        print(word[1:] + word[0] + 'ay')
```

```
Enter a sentence:  this is a test
```

```
histay
isway
away
esttay
```

[84]:
```
# more explicit version

sentence = input('Enter a sentence: ').strip()  # get input from the user
words = sentence.split()                        # create a list from user input

for word in words:        # iterate over each element of the list
    if word[0] in 'aeiou':
        print(word + 'way')
    else:
        print(word[1:] + word[0] + 'ay')
```

```
Enter a sentence:  this is another test for you to see and enjoy
```

```
histay
isway
anotherway
esttay
orfay
ouyay
otay
eesay
andway
enjoyway
```

19 `strip` vs `split`

The `str.strip` method: - Removes whitespace (space, tab, newline, carriage return, and vertical tab) in any quantity from *outside* of the string, on its edges. It doesn't do anything with whitespace inside. - It returns a new string, based on the original one.

The `str.split` method: - Always returns a new list of strings - The list is the result of breaking up the original string along the delimiter - By default, the delimiter is any whitespace (space, tab, newline, carriage return, and vertical tab) in any quantity.

It's true that if you're going to run `split` with the default, then you don't really need to use `strip` first.

[85]:
```
# another option

words = input('Enter a sentence: ').strip().split()

for word in words:        # iterate over each element of the list
    if word[0] in 'aeiou':
        print(word + 'way')
    else:
        print(word[1:] + word[0] + 'ay')
```

```
Enter a sentence:  this is a test
```

```
histay
isway
away
esttay
```

[87]:
```
words
```

[87]:
```
['this', 'is', 'a', 'test']
```

[88]:
```
words[0]
```

[88]:
```
'this'
```

[89]:
```
words[0][1]  # this means: return index 1 from words[0]
```

[89]:
```
'h'
```

20 Next up

- Joining strings together
- Tuples
- Unpacking

[90]:
```
# what if we have a list of strings, and we want to turn it into a new string?
```

```
# that is, I have something like this:

mylist = ['abcd', 'ef', 'ghij']

# and I want to get back 'abcd*ef*ghij'

# for this, we have the str.join method
# we run the method on the string that we want *between* the elements (the␣
 ↪"glue")

'*'.join(mylist)   # this will return a new string containing the elements of␣
 ↪mylist, glued together with '*'
```

[90]:
```
'abcd*ef*ghij'
```

[91]:
```
# what I can use as the glue? ANY STRING I want!

'\n'.join(mylist)
```

[91]:
```
'abcd\nef\nghij'
```

[92]:
```
print('\n'.join(mylist))
```

```
abcd
ef
ghij
```

[93]:
```
# what if I have a list of integer?

mylist = [10, 20, 30]

'*'.join(mylist)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[93], line 5
      1 # what if I have a list of integer?
      3 mylist = [10, 20, 30]
----> 5 '*'.join(mylist)

TypeError: sequence item 0: expected str instance, int found
```

[94]:
```
# it can actually be any iterable of strings
# meaning: any data that knows how to behave in a for loop

'*'.join('abcd')
```

[94]:
```
'a*b*c*d'
```

[95]:
```
s = 'this     is a      bunch    of words'
s.split()
```

[95]:
```
['this', 'is', 'a', 'bunch', 'of', 'words']
```

[96]:
```
' '.join(s.split())
```

[96]:
```
'this is a bunch of words'
```

[97]:
```
# empty string with join? squish all of the strings together
''.join(s.split())
```

[97]:
```
'thisisabunchofwords'
```

[98]:
```
# it's very common, in programs, to create lists, append to them repeatedly,␣
  ↪and then
# (at the end) to join them together

output = []
output.append('a')
output.append('b')
output.append('c')

print(' '.join(output))
```

```
a b c
```

21 Exercise: Pig Latin sentence (aesthetic edition)

Modify the previous exercise, such that the output is all printed on a single line. (Hint: Use lists and join to do this.)

[101]:
```
output = []    # start with an empty list for output
s = input('Enter a sentence: ').strip()

for word in s.split():
    if word[0] in 'aeiou':
        output.append(word + 'way')
    else:
        output.append(word[1:] + word[0] + 'ay')

print(' '.join(output))
```

```
Enter a sentence:  this is a test
```

```
histay isway away esttay
```

22 Tuples

Tuples are Python's equivalents to records or structs. They are the third "sequence" data type in Python.

- Strings
  - Can contain characters
  - Immutable
- Lists
  - Can contain anything at all
  - Mutable
- Tuples
  - Can contain anything at all
  - Immutable

You *can* think of tuples as immutable lists, or lists that cannot be changed. But the Python community doesn't want you to think that way:

- Use tuples if you have different types of values in your sequence. But if all of the values are of the same type, then you should use a list.

[102]:
```
# define a tuple using (usually) parentheses

t = (10, 20, 30, 40, 50)
type(t)
```

[102]:
```
tuple
```

[103]:
```
t = (10, 20)
type(t)
```

[103]:
```
tuple
```

[104]:
```
t = (10)     # this is very surprising to many people!
type(t)      # instead, you have to use
```

[104]:
```
int
```

[ ]:
```
t = (10,)    # notice the comma? That tells Python it's a tuple, and not an␣
↪integer
```

[105]:
```
t = ()
type(t)
```

[105]:
```
tuple
```

[106]:
```
# everything we can do with a string/list, we can do with a tuple, too

10 in t
```

[106]: False

[107]:
```
t = (10, 20, 30, 40, 50)

10 in t
```

[107]: True

[108]:
```
60 in t
```

[108]: False

[109]:
```
for one_item in t:
    print(one_item)
```

```
10
20
30
40
50
```

[110]:
```
# you don't really need parentheses to define one

x = 10, 20, 30   # on the right? that's a tuple
x
```

[110]: (10, 20, 30)

[111]:
```
# unpacking

mylist = [10, 20, 30]

x = mylist
x
```

[111]: [10, 20, 30]

[112]:
```
# what if I do this instead?

x,y,z = mylist  # the three elements of mylist were assigned to three variables!
x
```

[112]: 10

[113]:
```
y
```

[113]: 20

[114]:
```
z
```

[114]: 30

[115]:
```
# the number of variables must equal the number of elements in the sequence
x,y = mylist
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[115], line 2
      1 # the number of variables must equal the number of elements in the␣
 ↪sequence
----> 2 x,y = mylist

ValueError: too many values to unpack (expected 2)
```

[116]:
```
w,x,y,z = mylist
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[116], line 1
----> 1 w,x,y,z = mylist

ValueError: not enough values to unpack (expected 4, got 3)
```

[117]:
```
# let's define a tuple with info about a person

p = ('Reuven', 'Lerner', 46)
```

[118]:
```
print(p[0])
```

```
Reuven
```

[119]:
```
# I can take this tuple, and extract its fields into variables

first_name, last_name, shoe_size = p
```

[120]:
```
for index, one_item in enumerate('abcd'):
    print(f'{index}: {one_item}')
```

```
0: a
1: b
2: c
3: d
```

[121]:
```
# what does enumerate *really* return?

for one_thing in enumerate('abcd'):
```

```
    print(one_thing)
```

```
(0, 'a')
(1, 'b')
(2, 'c')
(3, 'd')
```

[122]:
```
# what if I just use t, for a tuple?

for t in enumerate('abcd'):
    print(t)
```

```
(0, 'a')
(1, 'b')
(2, 'c')
(3, 'd')
```

[123]:
```
# let's use unpacking!

for t in enumerate('abcd'):
    index, one_character = t    # unpacking
    print(f'{index}: {one_character}')
```

```
0: a
1: b
2: c
3: d
```

[124]:
```
# now let's use special for-loop syntax for unpacking

for index, one_character in enumerate('abcd'):
    print(f'{index}: {one_character}')
```

```
0: a
1: b
2: c
3: d
```

23 Exercise: Odds and evens

1. Define two empty lists, `odds` and `evens`.
2. Ask the user (repeatedly) to enter a string containing integers, separated by whitespace
   - If the user enters an empty string, break out of the loop
3. Go through each "word" in the string.
   - If it's not `isdigit`, then scold the user and move on.
4. For each "word" that contains numbers, turn it into an integer, and check if it's even or odd
   - Run `n % 2` (where `n` is the number) – if it returns 1, then the number is odd. Otherwise, even.
5. Print `odds` and `evens`

- We're asking the user to enter an unknown number of integers, in an unknown number of strings
- We'll need a `for` loop inside of the `while` loop
- We need to check for an empty string
- We `append` numbers to either `odds` or `evens`.

[126]:
```
odds = []
evens = []

while True:
    s = input('Enter numbers, separated by spaces: ').strip()

    if s == '':   # this is my chance to escape
        break

    for one_element in s.split():
        if one_element.isdigit():
            n = int(one_element)
            if n % 2 == 1:   # odd!
                odds.append(n)
            else:
                evens.append(n)

            print(one_element)
        else:
            print(f'Not numeric!')
            continue   # continue onto the next iteration of the for loop
```

```
Enter numbers, separated by spaces:  10 20 30 15 16 1a

10
20
30
15
16
Not numeric!

Enter numbers, separated by spaces:  2 3 4 5

2
3
4
5

Enter numbers, separated by spaces:
```

[128]:
```
print(evens)
print(odds)
```

```
[10, 20, 30, 16, 2, 4]
[15, 3, 5]
```

24 Next week

- Dictionaries
- Files