Lecture 7 - Iteration and Strings

Week 4 Monday

Miles Chen, PhD

Adapted from Chapter 7 and Chapter 8 of Think Python by Allen B Downey

Additional content on strings adapted from "Whirlwind Tour of Python" by Jake VanderPlas

Reassignment and Object ID

```
In [1]: # We can assign the value 5 to x
        x = 5
        x

Out[1]: 5
```

```
In [2]: # we can assign the value 7 to x
        # this changes the value of x
        x = 7
        x

Out[2]: 7
```

```
In [3]: # assign 5 to a
        a = 5
        a

Out[3]: 5

In [4]: # assign a to b
        b = a

In [5]: # when I assign 3 to a, the object a now points to a different object
        a = 3

In [6]: a

Out[6]: 3

In [7]: b

Out[7]: 5
```

Mutable and Immutable Objects

In Python, there are mutable and immutable objects.

Mutable objects are objects whose values can be changed.

Immutable objects are objects whose values cannot be changed.

Booleans, integers, floats, strings, and tuples are immutable.

Lists, dictionaries, and some other objects are mutable.

The integer 1 is an immutable object. We cannot change it to another value.

When I assign 1 to the object with name "a", I am associating the name a with the location in memory that stores the value 1. This is reflected with `id()`

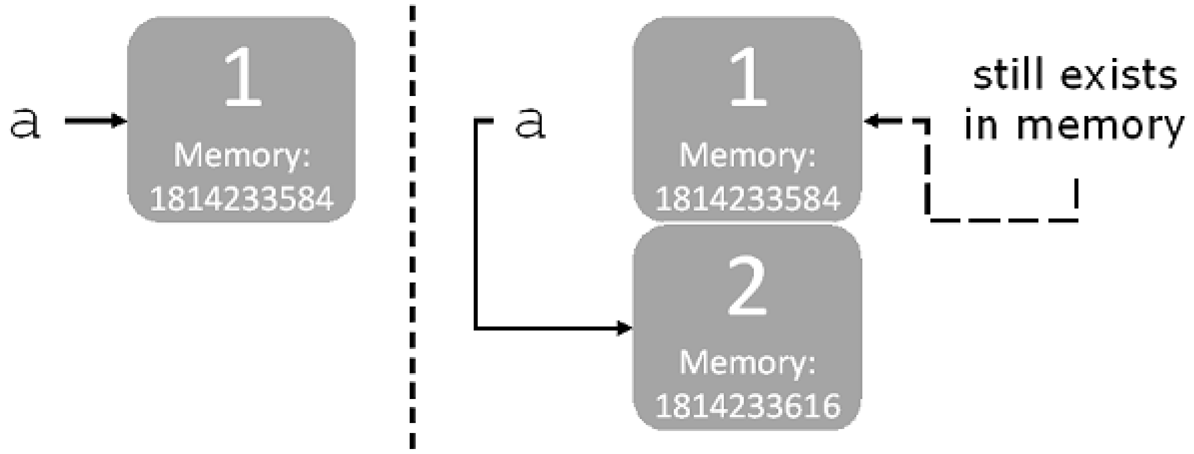
```
In [8]: a = 1
        id(a)
```

```
Out[8]: 140717332341136
```

When I change the value of `a` to something different, I am not changing the value of the integer, I am changing which object in memory that the name references.

```
In [9]: a = 2
        id(a)
```

```
Out[9]: 140717332341168
```



Img taken from: https://freecontent.manning.com/mutable-and-immutable-objects/ (https://freecontent.manning.com/mutable-and-immutable-objects/)

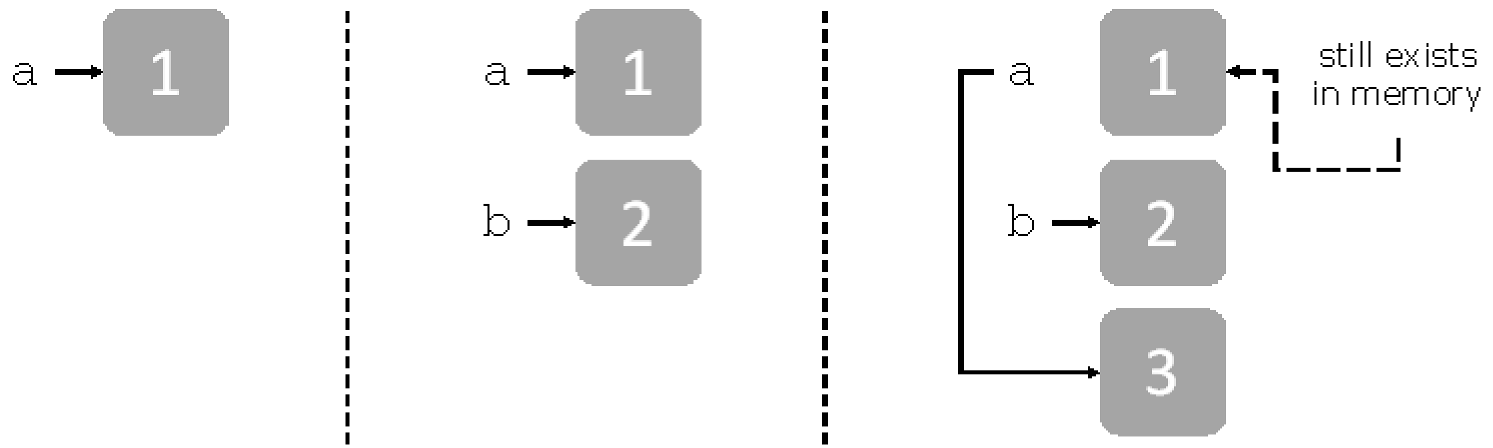
Memory Ids might be different in this image.

Objects of these types can't be modified once created.

Suppose you have the following lines of code, which are executed in the order shown. You initialize two variables, a and b, to two different objects with values 1 and 2, respectively.

Then you change the binding of variable a, to a different object with a value of 3

In [10]:
```
a = 1
b = 2
a = 3
```



Img taken from: [https://freecontent.manning.com/mutable-and-immutable-objects/ (https://freecontent.manning.com/mutable-and-immutable-objects/)](https://freecontent.manning.com/mutable-and-immutable-objects/)

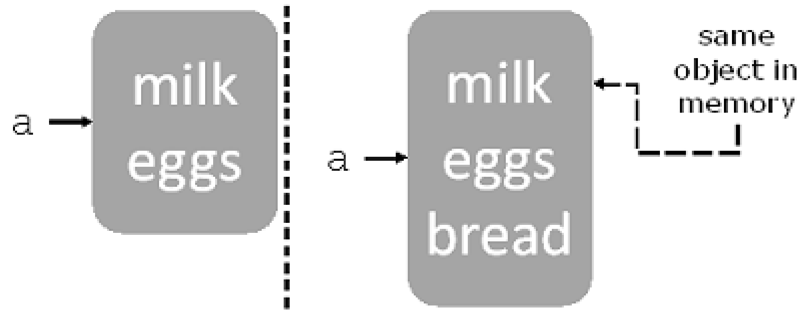
Memory Ids might be different in this image.

Once an immutable object loses its variable handle, the Python interpreter may delete the object to reclaim the computer memory it took up, and use it for something else.

Unlike some other programming languages, you (as the programmer) don't have to worry about deleting old objects – Python takes care of this for you through a process called "garbage collection.

Mutable objects can be modified

lists are mutable



In [11]:
```
# create a list and assign it to a
a = ["milk", "eggs"]
id(a)
```
Out[11]:
```
1983169574024
```
In [12]:
```
# creating a new list and binding it to a will change the id. This list is entirely sepa
rate of the old one.
a = ["milk", "eggs", "bread"]
id(a)
```
Out[12]:
```
1983170188872
```

```
In [13]: # We recreate a list and assign it to a
         a = ["milk", "eggs"]
         id(a)

Out[13]: 1983170309832
```

```
In [14]: # append modifies the list
         a.append("bread")
         a

Out[14]: ['milk', 'eggs', 'bread']
```

```
In [15]: # we see the id of the list is unchanged. We did not create a new list
         id(a)

Out[15]: 1983170309832
```

```
In [16]: id(a)

Out[16]: 1983170309832

In [17]: # this also modifies the list
         a[0] = "whole milk"

In [18]: a

Out[18]: ['whole milk', 'eggs', 'bread']

In [19]: id(a)

Out[19]: 1983170309832
```

Updating Values

```
In [20]: # we must initialize a value first before we can start to update it
         x = 1
         x

Out[20]: 1

In [21]: id(x)

Out[21]: 140717332341136

In [22]: x = x + 1
         x

Out[22]: 2

In [23]: # updating the value to a new integer changes its id
         id(x)

Out[23]: 140717332341168
```

`while` loops

In a `while` loop, the associated lines will run iteratively until the expression in the `while` statement is no longer `True` .

If the expression in the `while` statement is always True, the loop will run forever (unless there is a `break`).

Like everything else in python, the lines are associated with the `while` statement via indentation.

In [24]:
```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Blastoff!') # only runs after the loop ends
```

In [25]:
```
countdown(4)
```

```
4
3
2
1
Blastoff!
```

`break`

The `break` statement will exit a loop.

In [26]:
```python
n = 1
while True:
    print(n)
    n = n + 1
    if n == 8:
        break
```

```
1
2
3
4
5
6
7
```

Newton’t Method for finding a square root

aka babylonian method

https://en.wikipedia.org/wiki/Newton%27s_method#Square_root_of_a_number
(https://en.wikipedia.org/wiki/Newton%27s_method#Square_root_of_a_number)

https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Babylonian_method
(https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Babylonian_method)

In [27]:

```
# newton's method for finding a square root
def my_sqrt(a, est = 1, epsilon = 1e-10):
    while True:
        print(est)
        new_est = (est + a/est) / 2
        if abs(new_est - est) < epsilon:
            break
        est = new_est
```

```
In [29]: my_sqrt(4)

1
2.5
2.05
2.000609756097561
2.0000000929222947
2.000000000000002
```

```
In [30]: my_sqrt(100)

1
50.5
26.24009900990099
15.025530119986813
10.840434673026925
10.032578510960604
10.000052895642693
10.000000000139897
10.0
```

`for` loops

In a `for` loop, the associated lines will run iteratively for each element in the iterable.

In [85]:
```python
fruit_names = ["apple", "banana", "orange"]
print("Let's get crazy for fruit!")
for name in fruit_names:
    print(name.capitalize() + "!")
    print("YEAH!!")
```

```
Let's get crazy for fruit!
Apple!
YEAH!!
Banana!
YEAH!!
Orange!
YEAH!!
```

Strings

A string is a sequence

```
In [31]: fruit = "bananas"
```

```
In [32]: fruit[0]  # Python is 0-indexed
```

```
Out[32]: 'b'
```

```
In [33]: fruit[1]
```

```
Out[33]: 'a'
```

```
In [34]: fruit[-1] # last letter
```

```
Out[34]: 's'
```

```
In [35]: fruit[1.5]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-bf9cc58e8398> in <module>
----> 1 fruit[1.5]

TypeError: string indices must be integers
```

`len()` tells you the length of a string

```
In [36]: len(fruit)

Out[36]: 7
```

Subsetting Strings and strings as iterables

You can subset and slice a string much like you would a list or tuple:

```
In [37]: s = 'abcdefghijklmnopqrstuvwxyz'
```

```
In [38]: s[4:9]
```

```
Out[38]: 'efghi'
```

```
In [39]: s[-6:]
```

```
Out[39]: 'uvwxyz'
```

```
In [40]: for x in s[0:5]:
             print(x + '!')
```

```
a!
b!
c!
d!
e!
```

Strings are immutable

This means that when you use a method on a string, it does not modify the string itself and returns a new string object.

```
In [41]: # strings are immutable. You cannot modify a string that has been created.
         s[0] = 'b'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-41-26412a6b3b58> in <module>
      1 # strings are immutable. You cannot modify a string that has been created.
----> 2 s[0] = 'b'

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

```
In [42]: 'b' + s[1:] # if i wanted the string where the first letter is now b
```

```
Out[42]: 'bbcdefghijklmnopqrstuvwxyz'
```

String Methods

In [43]:
```python
name = "STATS 21 python and other technologies for data science"
print(name.upper())
print(name.capitalize()) # first character is capitalized
print(name.title())     # first character of each word is capitalized
print(name.lower())
print(name) # string itself is not modified
```

```
STATS 21 PYTHON AND OTHER TECHNOLOGIES FOR DATA SCIENCE
Stats 21 python and other technologies for data science
Stats 21 Python And Other Technologies For Data Science
stats 21 python and other technologies for data science
STATS 21 python and other technologies for data science
```

Count how many times a letter appears

```
In [44]: count = 0
         for letter in name:
             if letter == "e":
                 count = count + 1
         print(count)

         5
```

```
In [45]: # can be achieved with a simple method:
         name.count("e")

Out[45]: 5
```

```
In [46]: name.index('A') # index of the first instance

Out[46]: 2

In [47]: name.endswith("k")

Out[47]: False

In [48]: name.endswith("e")

Out[48]: True

In [49]: name.startswith("s")  # case sensitive

Out[49]: False
```

```
In [50]: # create multi-line strings with triple quotes
         name2 = '''   miles chen


         '''
         print(name2)

            miles chen
```

```
In [51]: name2.strip()  # removes extra whitespace

Out[51]: 'miles chen'
```

```
In [52]: name2 # remember strings are immutable, the original string still has the white space

Out[52]: '   miles chen \n\n\n'
```

string.split()

```
In [53]: name2.split() # the result of split() is a list
```

```
Out[53]: ['miles', 'chen']
```

```
In [54]: num_string = "2,3,4,7,8"
         print(num_string.split()) # defaults to splitting on space
         print(num_string.split(','))
```

```
['2,3,4,7,8']
['2', '3', '4', '7', '8']
```

```
In [55]: # list comprehension (covered later) to convert the split strings into int
         [int(x) for x in num_string.split(',')]

Out[55]: [2, 3, 4, 7, 8]
```

```
In [56]: # the list comprehension is a more concise version of the following code
         l = []
         for x in num_string.split(','):
             l.append(int(x))
         l

Out[56]: [2, 3, 4, 7, 8]
```

In [57]:
```python
print(name)
print(name.isalpha()) # has spaces and digits, so it is not strictly alpha
name3 = "abbaAZ"
name3.isalpha()
```

```
STATS 21 python and other technologies for data science
False
```

Out[57]:
```
True
```

In [58]:
```python
name4 = "abbaAZ4"
name4.isalpha()
```

Out[58]:
```
False
```

In [59]:
```python
# strings can span multiple lines with triple quotes
long_string = """Lyrics to the song Hallelujah
Well I've heard there was a secret chord
That David played and it pleased the Lord
But you don't really care for music, do you?"""
shout = long_string.upper()
print(shout)
word_list = long_string.split() # separates at spaces
print(word_list)
```

```
LYRICS TO THE SONG HALLELUJAH
WELL I'VE HEARD THERE WAS A SECRET CHORD
THAT DAVID PLAYED AND IT PLEASED THE LORD
BUT YOU DON'T REALLY CARE FOR MUSIC, DO YOU?
['Lyrics', 'to', 'the', 'song', 'Hallelujah', 'Well', "I've", 'heard', 'there', 'wa
s', 'a', 'secret', 'chord', 'That', 'David', 'played', 'and', 'it', 'pleased', 'the',
'Lord', 'But', 'you', "don't", 'really', 'care', 'for', 'music,', 'do', 'you?']
```

```
In [60]: long_string.splitlines() # separates at line ends
         # you'll notice that python defaults to using single quotes, but if the string contains
          an apostrophe,
         # it will use double quotes
```

```
Out[60]: ['Lyrics to the song Hallelujah',
          "Well I've heard there was a secret chord",
          'That David played and it pleased the Lord',
          "But you don't really care for music, do you?"]
```

```
In [61]: long_string.count("e")

Out[61]: 15
```

Searching for a letter

```
long_string = """Lyrics to the song Hallelujah
Well I've heard there was a secret chord
That David played and it pleased the Lord
But you don't really care for music, do you?"""
```

```
In [62]: def myfind(string, letter):
             index = 0
             while index < len(string):
                 if string[index] == letter:
                     return index
                 index = index + 1
             return -1
```

```
In [63]: myfind(long_string, "t")

Out[63]: 7
```

```
In [64]: # Python already has a find method built in
         long_string.find("t") # index of the first instance of 't'

Out[64]: 7

In [65]: long_string.index('t') # string.index() and string.find() are similar.

Out[65]: 7

In [66]: long_string.find('$') # string.find() returns a -1 if the character doesn't exist in the
         string

Out[66]: -1

In [67]: long_string.index('$')  # string.index() returns error if the character doesn't exist in
         the string.

         ---------------------------------------------------------------------------
         ValueError                                Traceback (most recent call last)
         <ipython-input-67-5b5715be5537> in <module>
         ----> 1 long_string.index('$')  # string.index() returns error if the character does
         n't exist in the string.

         ValueError: substring not found
```

`in` operator

returns a boolean value if the first string is a substring of the second string.

```
In [68]: 'a' in 'bananas'

Out[68]: True

In [69]: 'nan' in 'bananas'

Out[69]: True

In [70]: 'bad' in 'bananas'

Out[70]: False
```

String comparisons

Use of `>` or `<` compares strings in alphabetical order.

```
In [71]: 'A' < 'B'

Out[71]: True

In [72]: 'a' < 'b'

Out[72]: True

In [73]: 'c' < 'b'

Out[73]: False
```

```
In [74]: # capital letters are 'less than' lower case letters
         'A' < 'a'

Out[74]: True

In [75]: 'Z' < 'a'

Out[75]: True
```

```
In [76]: # digits are less than capital letters
         '1' < 'A'

Out[76]: True

In [77]: '0' < '1'

Out[77]: True

In [78]: '0' < '00'

Out[78]: True

In [79]: # must treat digits like "letters" with alphabetical rules
         '11' < '101'

Out[79]: False

In [80]: '!' < '@' # the sorting of symbols feels very arbitrary

Out[80]: True
```

In [81]:
```
# sorted order
string = '!@#$%^&*()[]{}\|;:,.<>/?1234567890ABCXYZabcxyz'
x = sorted(string)
print(x)
```

```
['!', '#', '$', '%', '&', '(', ')', '*', ',', '.', '/', '0', '1', '2', '3', '4', '5',
'6', '7', '8', '9', ':', ';', '<', '>', '?', '@', 'A', 'B', 'C', 'X', 'Y', 'Z', '[',
'\\', ']', '^', 'a', 'b', 'c', 'x', 'y', 'z', '{', '|', '}']
```