MODULE – 2 - Chapter 5. Data Types (Pg 91– 129)

Strings *(Pg 91 – 103):* Working with strings as single things, working with the parts of a string, Length, Traversal and the for loop, Slices, String comparison, Strings are immutable, the in and not in operators, A find function, Looping and counting, Optional parameters, The built-in find method, The split method, Cleaning up your strings, The string format method.

Tuples *(Pg 107 – 109):* Tuples are used for grouping data, Tuple assignment, Tuples as return values

Lists *(Pg 110 - 124):* List values, accessing elements, List length, List membership, List operations, List slices, Lists are mutable, List deletion, Objects and references, Aliasing, cloning lists, Lists and for loops, List parameters, List methods, Pure functions and modifiers, Functions that produce lists, Strings and lists, list and range, Nested lists, Matrices.

Dictionaries *(Pg 126 – 129):* Dictionary operations, dictionary methods, aliasing and copying.

Chapters: 5.1, 5.2, 5.3,5.4

==========

Chapter 5.1. Strings (Pg 91– 129)

Working with strings as single things, working with the parts of a string, Length, Traversal and the for loop, Slices, String comparison, Strings are immutable, the in and not in operators, A find function, Looping and counting, Optional parameters, The built-in find method, The split method, Cleaning up your strings, The string format method.

==========

A compound data type

- So far we have seen built-in types like ***int, float, bool, str*** and we've seen ***lists*** and ***pairs***.
- ***Strings, lists,*** and ***pairs*** are ***qualitatively different from the others*** because they are ***made up of smaller pieces***.
- In the case of strings, they're made up of smaller strings each containing one character.
- ***Types that comprise smaller pieces are called compound data types***. Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts. This ambiguity is useful.

Working with strings as single things

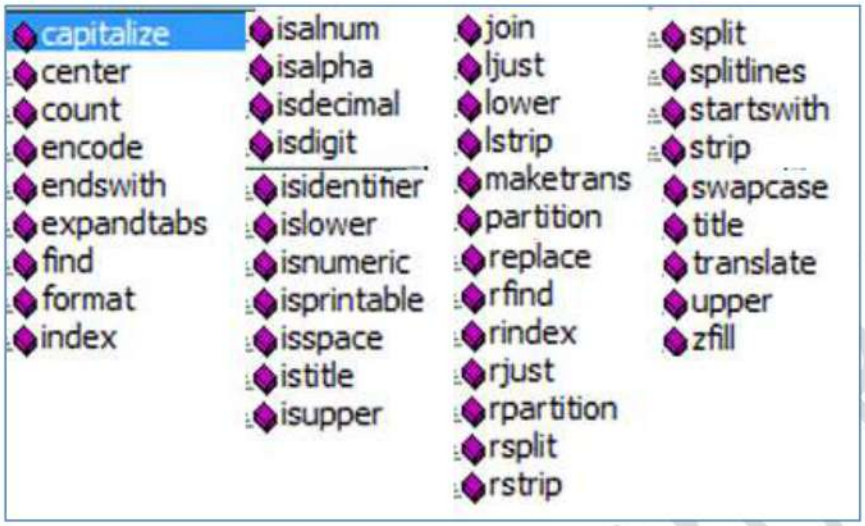
- A ***string is an object***, so ***each string instance has its own attributes and methods***.
- For example,

```
>>> our_string = "Hello, World!"
>>> all_caps = our_string.upper()
>>> all_caps
'HELLO, WORLD!'
```

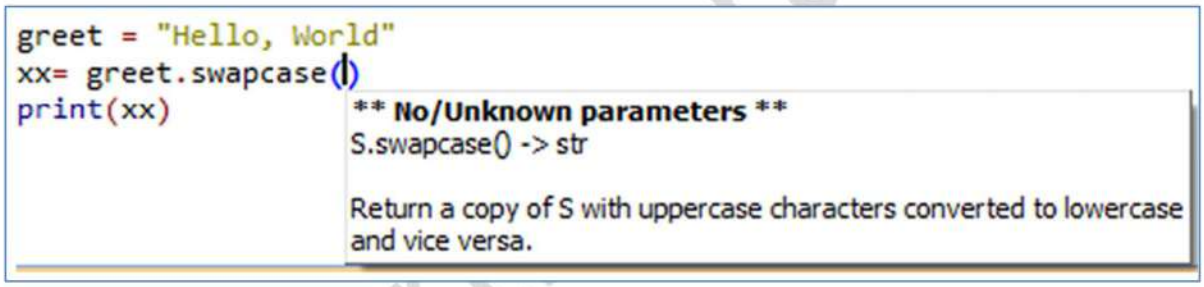
- ***upper*** is a ***method*** that can be ***invoked on any string object to create a new string***, in which all the characters are in uppercase. (The ***original string our_string remains unchanged***.)
- There are also methods such as ***lower***, ***capitalize***, and ***swapcase*** that do other interesting stuff.
- Type the following into an editor like ***Spyder*** or ***PyScripter*** script:

```
our_string = "Hello, World!"
new_string = our_string.
```

- When you type the period to select one of the methods of our_string, your editor might pop up a selection window —typically by pressing Tab — showing all the methods that could be used on your string.



- When you type the name of the method, some further help about its parameter and return type, and its docstring, may be displayed by your scripting environments (for instance, in a Jupyter notebook you can get this information by pressing Shift+Tab after a function name).



Working with the parts of a string

- The **indexing operator** (Python uses square brackets to enclose the index) selects a single character substring from a string:

```
>>> fruit = "banana"
>>> letter = fruit[1]
>>> print(letter)
a
```

- The **expression fruit[1]** selects character number 1 from fruit, and creates a new string containing just this one character. The variable letter refers to the result. When we display letter, we could get a surprise.
- Computer scientists always start counting from zero! The letter at subscript position zero of **"banana"** is **b**. So at **position [1]** we have the letter **a**. If we want to access the zero-eth letter of a string, we just place 0, or any expression that evaluates to 0, in between the brackets:

```
>>> letter = fruit[0]
>>> print(letter)
B
```

- The **expression** in brackets is called an **index**. An **index** specifies a ***member of an ordered collection***, in this case the collection of characters in the string.
- The index indicates which one you want, hence the name. It can be any integer expression.

- We can use **enumerate** to visualize the indices:

```
>>> fruit = "banana"
>>> list(enumerate(fruit))
[(0, 'b'), (1, 'a'), (2, 'n'), (3, 'a'), (4, 'n'), (5, 'a')]
```

- Note that indexing returns a string — ***Python has no special type for a single character***. It is just a string of length 1. The same indexing notation works to extract elements from a list:

```
>>> prime_numbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
>>> prime_numbers[4]
11
>>> friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
>>> friends[3]
'Angelina'
```

Length

- The **"len"** function, when applied to a string, returns the number of characters in a string:

```
>>> word = "banana"
>>> len(word)
6
```

- To get the last letter of a string, you might be tempted to try something like this:

```
size = len(word)
last = word[size]      # ERROR!
```

- That won't work. It causes the runtime error ***IndexError: string index out of range***. The reason is that there is no character at ***index position 6 in "banana".*** Because we ***start counting at zero***, the ***six indexes are numbered 0 to 5***. To get the last character, we have to subtract 1 from the length of word:

```
size = len(word)
last = word[size-1]
```

- Alternatively, we can use negative indices, which count backward from the end of the string.
- The expression ***word[-1] yields the last letter***, ***word[-2] yields the second to last***, and so on. As you might have guessed, indexing with a negative index also works like this for lists.

Traversal and the "for" loop

- A lot of computations involve ***processing a string one character at a time***. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This ***pattern of processing*** is called a ***traversal***.
- One way (a very bad way) to encode a traversal is with a ***"while"*** statement and how the ***"for"*** loop can easily iterate over the elements in a list and it can do so for strings as well:

```
ix = 0
while ix < len(fruit):
    letter = fruit[ix]
    print(letter)
    ix += 1
```

```
word="Banana"
for letter in word:
    print(letter)
```

- This ***"while"*** loop traverses the string and displays each letter on a line by itself. It uses ***"ix"*** for the index, which does not make it any clearer. The loop condition is ***ix < len(fruit),*** so when ***"ix"*** is equal to the length of the string, the condition is false, and the body of the loop is not executed.

The last character accessed is the one with the index ***len(fruit)-1***, which is the last character in the string. However, this code is a lot longer than it needs to be, and not very clear at all.

- Each time through the ***"for"*** loop, the next character in the string is assigned to the variable "letter" ***(Error in text book 'c' – Page 94).*** The loop continues until no characters are left. Here we can see the expressive power the ***"for"*** loop gives us compared to the while loop when traversing a string.
- The following example shows how to use concatenation and a ***"for"*** loop to generate ***anabecedarian*** series. ***Abecedarian refers to a series or list in which the elements appear in alphabetical order.*** For example, in Robert McCloskey's book Make Way for Ducklings, the names of the ducklings are Jack, Kack, Lack, Mack and Nack. This loop outputs these names in order:

```
prefixes = "JKLMN"
suffix = "ack"

for p in prefixes:
    print(p + suffix)
```

The output:

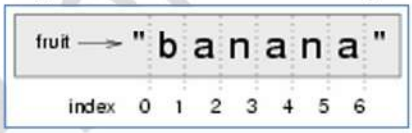
```
Jack
Kack
Lack
Mack
Nack
```

Slices

- A ***sub string of a string*** is obtained by taking a ***"slice"***. Similarly, we can slice a list to refer to some ***sub list of the items*** in the list:

```
>>> phrase = "Pirates of the Caribbean"
>>> print(phrase[0:7])
Pirates
>>> print(phrase[11:14])
the
>>> print(phrase[13:24])
e Caribbean
>>> friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki"]
>>> print(friends[2:4])
['Brad', 'Angelina']
```

- The operator ***[n:m]*** returns the part of the string from the ***n'th*** character to the ***m'th*** character, ***including the first*** but ***excluding the last***. This behavior makes sense if you imagine the indices pointing between the characters, as in the following diagram:



- If you imagine this as a piece of paper, the slice operator ***[n:m]*** copies out the part of the paper between the ***"n"*** and ***"m"*** positions. Provided ***"m"*** and ***"n"*** are both within the bounds of the string, your result will be of ***length (m-n).***
- Three tricks are added to this:
  1. if you omit the first index (before the colon), the slice starts at the beginning of the string (or list).
  2. If you omit the second index, the slice extends to the end of the string (or list).

3. Similarly, if you provide value for n that is bigger than the length of the string (or list), the slice will take all the values up to the end. (It won't give an "out of range" error like the normal indexing operation does.) Thus:

```
>>> word = "banana"
>>> word[:3]
'ban'
>>> word[3:]
'ana'
>>> word[3:999]
'ana'
```

- ***What do you think phrase[:] means? What about friends[4:]? phrase[-5:-3]?***

Expression	Meaning	Example	Output
`phrase[:]`	Returns the **entire sequence**	`phrase = "Python Programming"` `print(phrase[:])`	
`friends[4:]`	Returns elements **from index 4 to the end**	`friends = ["Ram", "Joe", "Ravi", "Anu", "Sarathy", "Meena"]` `print(friends[4:])`	
`phrase[-5:-3]`	Returns characters **from 5th from end to 3rd from end (excluding -3)**	`phrase = "Programming"` `print(phrase[-5:-3])`	

String comparison

- The comparison operators work on strings. To see if two strings are equal:

```
if word == "banana":
    print("Yes, we have no bananas!")
```

Other comparison operations are useful for putting words in *lexicographical* order:

```
if word < "banana":
    print("Your word, " + word + ", comes before banana.")
elif word > "banana":
    print("Your word, " + word + ", comes after banana.")
else:
    print("Yes, we have no bananas!")
```

- This is similar to the ***alphabetical order*** you would ***use with a dictionary***, except that all the uppercase letters come before all the lowercase letters. As a result:

  Your word, Zebra, comes before banana.
- A ***common way to address this problem is to convert strings to a standard format***, such as ***all lowercase, before performing the comparison***.
- A more ***difficult problem*** is making the program realize that ***zebras*** are ***not fruit***.

Strings are immutable

- It is ***tempting to use the [] operator on the left side of an assignment***, with the intention of changing a character in a string. For example:

```
greeting = "Hello, world!"
greeting[0] = 'J'          # ERROR!
print(greeting)
```

- Instead of producing the output ***Jello, world!,*** this code ***produces the runtime error*** ***TypeError: 'str' object does not support item assignment.***
- ***Strings are immutable***, which means ***you can't change an existing string***. The ***best you can do*** is ***create a new string*** that is a variation on the original:

```
greeting = "Hello, world!"
new_greeting = "J" + greeting[1:]
print(new_greeting)
```

- The solution here is to concatenate a new first letter onto a slice of greeting. This operation has no effect on the original string.

---

the "in" and "not in" operators

- The ***"in"*** operator tests for membership. When both of the arguments to ***"in"*** are strings, ***"in" checks whether the left argument is a substring of the right argument.***

>>> "p" in "apple"
True
>>> "i" in "apple"
False
	>>>"ap" in "apple"
	True
	>>>"pa" in "apple"
	False

- Note that a string is a substring of itself, and the empty string is a substring of any other string. (Also note that computer scientists like to think about these edge cases quite carefully!)

>>>"a" in "a" True >>>"apple" in "apple" True	>>>"" in "a" True >>>"" in "apple" True

- The ***"not in"*** operator returns the logical opposite results of ***"in"***:

```
>>>"x" not in "apple"
True
```

- Combining the ***"in"*** operator with string concatenation using ***"+"***, we can write a function that removes all the vowels from a string:

```
def remove_vowels(phrase):
    vowels = "aeiou"
    string_sans_vowels = ""
    for letter in phrase:
        if letter.lower() not in vowels:
            string_sans_vowels += letter
    return string_sans_vowels
```

- Important to note is the ***letter.lower()*** in line 5, without it, any upper case vowels would not be removed.

A "find" function

- What does the following function do?

```
def my_find(haystack, needle):
    for index, letter in enumerate(haystack):
        if letter == needle:
            return index
    return -1
haystack = "Bananarama!"
print(haystack.find('a'))
print(my_find(haystack, 'a'))
```

- In a sense, ***"find"*** is the ***opposite of the indexing operator***. Instead of taking an index and extracting the corresponding character, ***it takes a character and finds the index where that character appears***. If the ***character is not found***, the function ***returns-1.***
- This is another example where we see a ***"return"*** statement inside a loop. If ***letter == needle***, the function returns immediately, breaking out of the loop prematurely.
- If the character doesn't appear in the string, then the program exits the loop normally and returns-1.
- This pattern of computation is sometimes called a ***eureka traversal*** or ***short-circuit evaluation***, because as soon as we find what we are looking for, we can cry "Eureka!", take the short-circuit, and stop looking.

Looping and counting

- The following program counts the number of times the letter ***"a"*** appears in a string, and is another example of the counter pattern introduced in ***"Counting digits":***

```
def count_a(text):
    count = 0
    for letter in text:
        if letter == "a":
            count += 1
    return(count)
print(count_a("banana") == 3)
```

OUTPUT
True

Optional parameters

- To find the locations of the second or third occurrence of a character in a string, we can modify the ***"find"*** function, adding a third parameter for the starting position in the search string:

```
def find2(haystack, needle, start):
    for index,letter in enumerate(haystack[start:]):
        if letter == needle:
            return index + start
    return -1
print(find2("banana", "a", 2) == 3)
```

- The call ***find2("banana", "a", 2)*** now returns ***3***, the index of the first occurrence of ***"a"*** in ***"banana"*** starting the ***search at index 2.*** What does ***find2("banana", "n", 3)*** return? If you said, 4, there is a good chance you understand how find2 works.
- We can combine ***"find"*** and ***"find2"*** using an optional parameter:

```
def find(haystack, needle, start=0):
    for index,letter in enumerate(haystack[start:]):
        if letter == needle:
            return index + start
    return -1
```

- When a function has an optional parameter, the caller ***may*** provide a matching argument. If the third argument is provided to ***find***, it gets assigned to ***start***. But if the caller leaves the argument out, then start is given a default value indicated by the assignment ***start=0*** in the function definition.
- So the call ***find("banana", "a", 2)*** to this version of find behaves just like ***find2***, while in the call ***find("banana", "a"),*** start will be set to the default value of ***0***.
- Adding another optional parameter to find makes it search from a starting position, up to but not including the end position:

```
def find(haystack, needle, start=0, end=-1):
    for index,letter in enumerate(haystack[start:end])
        if letter == needle:
            return index + start
    return -1
```

- The semantics of start and end in this function are precisely the same as they are in the ***"range"*** function.

The built-in "find" method

- Now that we've done all this work to write a powerful ***"find"*** function, we can reveal that strings already have their own built-in find method. It can do everything that our code can do, and more!
- The built-in find method is more general than our version. It can find substrings, not just single characters:

```
>>> "banana".find("nan")
2
>>> "banana".find("na", 3)
4
```

- Usually we'd prefer to use the methods that Python provides rather than reinvent our own equivalents. But many of the built-in functions and methods make good teaching exercises, and the underlying techniques you learn are your building blocks to becoming a proficient programmer.

The "split" method

- One of the most useful methods on strings is the split method: it splits a single multi-word string into a list of individual words, removing all the whitespace between them (Whitespace means any tabs, newlines, or spaces.)
- This allows us to read input as a single string and split it into words.

```
>>> phrase = "Well I never did said Alice"
>>> words = phrase.split()
>>> words
['Well', 'I', 'never', 'did', 'said', 'Alice']
```

Cleaning up your strings

- We'll often work with strings that contain punctuation, or tab and newline characters.
- If we're writing a program, say, to count word frequencies or check the spelling of each word, we'd prefer to strip off these unwanted characters.
- We'll show just one example of how to strip punctuation from a string. Remember that strings are immutable, so we cannot change the string with the punctuation — we need to traverse the original string and create a new string, omitting any punctuation:

```
punctuation = "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~"
def remove_punctuation(phrase):
    phrase_sans_punct = ""
    for letter in phrase:
        if letter not in punctuation:
            phrase_sans_punct += letter
    return phrase_sans_punct
```

- Setting up that first assignment is messy and error-prone. Fortunately, the Python ***string*** module already does it for us. So, we will make a slight improvement to this program—we'll import the ***string*** module and use its definition:

```
import string
def remove_punctuation(phrase):
    phrase_sans_punct = ""
    for letter in phrase:
        if letter not in string.punctuation:
            phrase_sans_punct += letter
    return phrase_sans_punct
```

- Try the examples below: "Well, I never did!", said Alice. "Are you very, very, sure?"
- Composing together this function and the split method from the previous section makes a useful combination— we'll clean out the punctuation, and split will clean out the new lines and tabs while turning the string into a list of words:

```
my_story = """
Pythons are constrictors, which means that they will 'squeeze' the life
out of their prey. They coil themselves around their prey and with
each breath the creature takes the snake will squeeze a little tighter
you guessed it --- snake POOP! """
words = remove_punctuation(my_story).split()
print(words)
```

The output:

```
['Pythons', 'are', 'constrictors', ... , 'it', 'snake', 'POOP']
```

The string "format" method

- The easiest and most powerful way to format a string in Python3 is to use the format method. To see how this works, let's start with a few examples:

```
phrase = "His name is {0}!".format("Arthur")
print(phrase)
name = "Alice"
age = 10
phrase = "I am {1} and I am {0} years old.".format(age, name)
print(phrase)
phrase = "I am {0} and I am {1} years old.".format(age, name)
print(phrase)
x = 4
y = 5
phrase = "2**10 = {0} and {1} * {2} = {3:f}".format(2**10, x, y, x * y)
print(phrase)
```

- Running the script produces:

```
His name is Arthur!
I am Alice and I am 10 years old.
I am 10 and I am Alice years old.
2**10 = 1024 and 4 * 5 = 20.000000
```

- The template string contains place holders, ... {0} ... {1} ... {2} ...etc.
- The *"format"* method substitutes its arguments into the place holders. The numbers in the place holders are indexes that determine which argument gets substituted — make sure you understand!
- Each of the replacement fields can also contain a format specification, this modifies how the substitutions are made into the template, and can control things like:
  - whether the field is aligned to the left
  - the width allocated to the field within the result string (a number like 10)
  - the type of conversion
  - if the type conversion is a float, you can also specify how many decimal places are wanted (typically, .2f is useful for working with currencies to two decimal places.)
- Let's do a few simple and common examples that should be enough for most needs. If you need to do anything more esoteric, use help and read all the powerful, gory details:

```
name1 = "Paris"
name2 = "Whitney"
name3 = "Hilton"
print("Pi to three decimal places is {0:.3f}".format(3.1415926))
print("|||{0:<15}|||{1:^15}|||{2:>15}|||Born in {3}|||"
      .format(name1,name2,name3,1981))
print("The decimal value {0} converts to hex value {0:x}"
      .format(123456))
```

output:

```
Pi to three decimal places is 3.142
|||Paris      |||    Whitney    |||     Hilton|||Born in 1981|||
The decimal value 123456 converts to hex value 1e240
```

- You can have multiple placeholders indexing the same argument, or perhaps even have extra arguments that are not referenced at all:

```
letter = """
Dear {0} {2}.
 {0}, I have an interesting money-making proposition for you!
"""
print(letter.format("Paris", "Whitney", "Hilton"))
print(letter.format("Bill", "Henry", "Gates"))
```

OUTPUT

```
Dear Paris Hilton.
 Paris, I have an interesting money-making proposition for you!

Dear Bill Gates.
 Bill, I have an interesting money-making proposition for you!
```

- As you might expect, you'll get an index error if your place holders refer to arguments that you do not provide:

```
>>> "hello {3}".format("Dave")
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
IndexError: tuple index out of range
```

- The following example illustrates ***the real utility of string formatting***. First, we'll try to print a table without using string formatting:

Python code	*OUTPUT*

```
print("i\ti**2\ti**3\ti**5")
for i in range(1,11):
    print(i,"\t",i**2,"\t\t",i**3,"\t\t",i**5)
```

```
i	i**2	i**3	i**5
1	1		1		1
2	4		8		32
3	9		27		243
4	16		64		1024
5	25		125		3125
6	36		216		7776
7	49		343		16807
8	64		512		32768
9	81		729		59049
10	100		1000		100000
```

- One possible solution would be to change the tab width, but the first column already has more space than it needs. The best solution would be to set the width of each column independently. String formatting provides a much nicer solution. This Python program prints a **formatted table of powers of numbers** from **1 to 10**.

```
layout = "{0:>4}{1:>6}{2:>6}{3:>8}"
print(layout.format("i", "i**2", "i**3", "i**5"))
for i in range(1, 11):
    print(layout.format(i, i**2, i**3, i**5))
```

{0:>4} - Print argument 0 right aligned in 4 spaces, {1:>6} - Print argument 1 right aligned in 6 spaces
{2:>6} Print argument 2 right aligned in 6 spaces, {3:>8} Print argument 3 right aligned in 8 spaces

- Running this version produces the following (much more satisfying) output:

```
 i   i**2   i**3    i**5
 1     1      1       1
 2     4      8      32
 3     9     27     243
 4    16     64    1024
 5    25    125    3125
 6    36    216    7776
 7    49    343   16807
 8    64    512   32768
 9    81    729   59049
10   100   1000  100000
```

Summary

indexing ([]) Access a single character in a string using its position (starting from 0). Example: `"This"[2]` evaluates to `"i"`.

length function (len) Returns the number of characters in a string. Example: `len("happy")` evaluates to `5`.

for loop traversal (for) *Traversing* a string means accessing each character in the string, one at a time. For example, the following for loop:

```
for ch in "Example":
    ...
```

executes the body of the loop 7 times with different values of `ch` each time.

slicing ([:]) A *slice* is a substring of a string. Example: `'bananas and cream'[3:6]` evaluates to `ana` (so does `'bananas and cream'[1:4]`).

string comparison (>, <, >=, <=, ==, !=) The six common comparison operators work with strings, evaluating according to *lexicographical* order. Examples: `"apple" < "banana"` evaluates to `True`. `"Zeta" < "Appricot"` evaluates to `False`. `"Zebra" <= "aardvark"` evaluates to `True` because all upper case letters precede lower case letters.

in and not in operator (in, not in) The `in` operator tests for membership. In the case of strings, it tests whether one string is contained inside another string. Examples: `"heck" in "I'll be checking for you."` evaluates to `True`. `"cheese" in "I'll be checking for you."` evaluates to `False`.

Chapter 5.2. Tuples (Pg 107– 109)

Tuples are used for grouping data, Tuple assignment, Tuples as return values

Tuples are used for grouping data

- We could group together pairs of values by surrounding with parentheses. Recall this example:
  >>> year_born = ("Paris Hilton", 1981)
- This is an example of a ***data structure*** — a mechanism for grouping and organizing data to make it easier to use. The pair is an example of a ***tuple***.
- Generalizing this, a tuple can be used to group any number of items into a single compound value. Syntactically, a tuple is a comma-separated sequence of values. Although it is not necessary, it is conventional to enclose tuples in parentheses:
  >>> julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")
- ***Tuples*** are ***useful for representing*** what other languages often ***call records (or structs)*** — some related information that belongs together, like your student record.
- A tuple lets us "chunk" together related information and use it as a single thing.

- Tuples support the same sequence operations as strings. The index operator selects an element from a tuple.

```
>>> julia[2]
1967
```

- But if we try to use item assignment to modify one of the elements of the tuple, we get an error:

```
>>> julia[0] = "X"
TypeError: 'tuple' object does not support item assignment
```

- So like strings, tuples are permanent. Once Python has created a tuple in memory, it cannot be changed.
- Of course, ***even if we can't modify the elements of a tuple, we can always make the julia variable reference a new tuple holding different information***.
- To construct the new tuple, it is convenient that we can slice parts of the old tuple and join up the bits to make the new tuple. So if julia has a new recent film, we could change her variable to reference a new tuple that used some information from the old one:

```
>>> julia = julia[:3] + ("Eat Pray Love", 2010) + julia[5:]
>>> julia ("Julia", "Roberts", 1967, "Eat Pray Love", 2010, "Actress", "Atlanta, Georgia")
```

- To create a tuple with a single element, we have to include the final comma, because without the final comma, Python treats the (5) below as an integer in parentheses:

```
>>> tup = (5,)
>>> type(tup)
<class 'tuple'>
>>> x = (5)
>>> type(x)
<class 'int'>
```

Tuple assignment

- Python has a very powerful tuple assignment feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment.

```
(name, surname, year_born, movie, year_movie, profession, birthplace) = julia
```

- This does the equivalent of seven assignment statements, all on one easy line. One requirement is that the number of variables on the left must match the number of elements in the tuple.
- One way to think of ***tuple assignment*** is as ***tuple packing/unpacking***.
- In ***tuple packing***, the ***values on the left are 'packed' together in a tuple***.

```
>>> bob = ("Bob", 19, "CS")  # tuple packing
```

- In ***tuple unpacking***, the values in a ***tuple on the right are 'unpacked' into the variables/names*** on the right:

```
>>> bob = ("Bob", 19, "CS")
>>> (name, age, studies) = bob        # tuple unpacking
>>> name
'Bob'
>>> age
19
>>> studies
'CS'
```

- Once in a while, it is useful to swap the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap a and b:

temp = a
a = b
b = temp

- Tuple assignment solves this problem neatly: ***(a, b) = (b, a)***
- The ***left side is a tuple of variables***; the ***right side is a tuple of values***.
- Each value is assigned to its respective variable.
- All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile.
- Naturally, the number of variables on the left and the number of values on the right have to be the same:

  >>> (one, two, three, four) = (1, 2, 3)
  ValueError: need more than 3 values to unpack

Tuples as return values

- ***Functions*** can always only ***return a single value***, but by making that value a ***tuple***, we can ***effectively group together as many values as we like and return them together.***
- This is very useful
  - we often want to know some batsman's highest and lowest score, or
  - we want to find the mean and the standard deviation, or
  - we want to know the year, the month, and the day, or
  - if we're doing some some ecological modelling we may want to know the number of rabbits and the number of wolves on an island at a given time.
- For example, we could write a function that returns both the area and the circumference of a circle of radius r:

```
def circle_stats(r):
    circumference = 2 * math.pi * r
    area = math.pi * r * r
    return (circumference, area)
```

Composability of Data Structures

- An example where one of the items in the tuple was itself a list:

```
students = [
    ("John", ["CompSci", "Physics"]),
    ("Vusi", ["Maths", "CompSci", "Stats"]),
    ("Jess", ["CompSci", "Accounting", "Economics", "Management"]),
    ("Sarah", ["InfSys", "Accounting", "Economics", "CommLaw"]),
    ("Zuki", ["Sociology", "Economics", "Law", "Stats", "Music"])]
```

- Tuples items can themselves be other tuples. For example, we could improve the information about our movie stars to hold the full date of birth rather than just the year, and we could have a list of some of her movies and dates that they were made, and so on:

```
julia_more_info = ( ("Julia", "Roberts"), (8, "October", 1967),
                    "Actress", ("Atlanta", "Georgia"),
                    [ ("Duplicity", 2009),
                      ("Notting Hill", 1999),
                      ("Pretty Woman", 1990),
                      ("Erin Brockovich", 2000),
                      ("Eat Pray Love", 2010),
                      ("Mona Lisa Smile", 2003),
                      ("Oceans Twelve", 2004) ])
```

- Notice in this case that the tuple has just five elements — but each of those in turn can be another tuple, a list, a string, or any other kind of Python value. This property is known as being ***heterogeneous***, meaning that ***it can be composed of elements of different types***.

Glossary

- ***data structure***: An organization of data for the purpose of making it easier to use.
- ***immutable data value***: A data value which cannot be modified. Assignments to elements or slices (sub-parts) of immutable values cause a runtime error.
- ***mutable data value***: A data value which can be modified. The types of all mutable values are compound types. Lists and dictionaries are mutable; strings and tuples are not.
- ***Tuple***: An immutable data value that contains related elements. Tuples are used to group together related data, such as a person's name, their age, and their gender.
- ***tuple assignment***: An assignment to all of the elements in a tuple using a single assignment statement. Tuple assign ment occurs simultaneously rather than in sequence, making it useful for swapping values.

5.3. Lists (Pg 110– 124)

List values, accessing elements, List length, List membership, List operations, List slices, Lists are mutable, List deletion, Objects and references, Aliasing, cloning lists, Lists and for loops, List parameters, List methods, Pure functions and modifiers, Functions that produce lists, Strings and lists, list and range, Nested lists, Matrices.

- A ***list*** is an ***ordered collection of values***. The values that make up a list are called its elements, or its items. We will use the term element or item to mean the same thing.
- Lists are similar to strings, which are ordered collections of characters, except that the elements of a list can be of any type.
- ***Lists and strings*** — and other collections that maintain the order of their items — ***are called sequences***.

List values

- There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

  numbers = [10, 20, 30, 40]

  words = ["spam", "bungee", "swallow"]
- The first example is a list of four integers. The second is a list of three strings.

- The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (amazingly) another list:

  stuffs = ["hello", 2.0, 5, [10, 20]]

- ***A list within another list is said to be nested***. Finally, a list with no elements is called an ***empty list***, and is ***denoted []***.
- We have already seen that we can assign list values to variables or Pass lists as parameters to function:

  >>> vocabulary = ["apple", "cheese", "dog"]
  >>> numbers = [17, 123]
  >>> an_empty_list = []
  >>> print(vocabulary, numbers, an_empty_list)
  ["apple", "cheese", "dog"] [17, 123] []

Accessing elements

- The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string — the index operator: [].
- The expression inside the brackets specifies the index. Remember that the indices start at 0:

  >>> numbers[0]
  17

- Any expression evaluating to an integer can be used as an index:

```
>>> numbers[9-8]
123
>>> numbers[1.0]
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: list indices must be integers, not float
```

- If you try to access or assign to an element that does not exist, you get a runtime error:

```
>>> numbers[2]
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
IndexError: list index out of range
```

- It is common (but wrong!) to use a loop variable as a list index.

```
horsemen = ["war", "famine", "pestilence", "death"]
for i in [0, 1, 2, 3]:
    print(horsemen[i])
```

Output

```
war
famine
pestilence
death
```

- Each time through the loop, the variable i is used as an index into the list, printing the i'th element. This pattern of computation is called a ***list traversal***.
- Need or use the index for anything besides getting the items from the list, so this more direct version — where the for loop gets the items — is much more clear!

```
horsemen = ["war", "famine", "pestilence", "death"]
for h in horsemen:
    print(h)
```

List length (len)

- The function ***"len"*** returns the ***length of a list***, which is ***equal to the number of its elements***. If you are going to use an integer index to access the list, it is a good idea to use this value as the upper bound of a loop instead of a constant. That way, if the size of the list changes, you won't have to go through the program changing all the loops; they will work correctly for any size list:

```
horsemen = ["war", "famine", "pestilence", "death"]
for i in range(len(horsemen)):
        print(horsemen[i])
```

- The last time the body of the loop is executed, i is len(horsemen)- 1, which is the index of the last element.

```
horsemen = ["war", "famine", "pestilence", "death"]
for horseman in horsemen:
        print horseman
```

- Although a list can contain another list, the nested list still counts as a single element in its parent list. The length of this list is 4:

```
>>> len(["car makers", 1, ["Ford", "Toyota", "BMW"], [1, 2, 3]])
4
```

List membership

- ***"in"*** and ***"not in"*** are ***Boolean operators that test membership in a sequence***. We used them previously with strings, but they also work with lists and other sequences:

```
>>> horsemen = ["war", "famine", "pestilence", "death"]
>>> "pestilence" in horsemen
True
>>> "debauchery" in horsemen
False
>>> "debauchery" not in horsemen
True
```

- Using this produces a more elegant version of the nested loop program we previously used to count the number of students doing Computer Science in the section ***Nested Loops for Nested Data***:

```
students = [
    ("John", ["CompSci", "Physics"]),
    ("Vusi", ["Maths", "CompSci", "Stats"]),
    ("Jess", ["CompSci", "Accounting", "Economics", "Management"]),
    ("Sarah", ["InfSys", "Accounting", "Economics", "CommLaw"]),
    ("Zuki", ["Sociology", "Economics", "Law", "Stats", "Music"])]
# Count how many students are taking CompSci
counter = 0
for name, subjects in students:
    if "CompSci" in subjects:
            counter += 1
print("The number of students taking CompSci is", counter)
```

List operations

The + operator concatenates lists:

```
>>> first_list = [1, 2, 3]
>>> second_list = [4, 5, 6]
>>> both_lists = first_list + second_list
>>> both_lists
[1, 2, 3, 4, 5, 6]
```

Similarly, the * operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats `[0]` four times. The second example repeats the list `[1,2,3]` three times.

List slices

- The slice operations we saw previously with strings let us work with sub lists:

```
>>> a_list = ["a","b","c","d","e","f"]
>>> a_list[1:3]
['b', 'c']
>>> a_list[:4]
['a', 'b', 'c', 'd']
>>> a_list[3:]
['d', 'e', 'f']
>>> a_list[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Lists are mutable

- Unlike strings, lists are mutable, which means we can change their elements. Using the index operator on the left side of an assignment, we can update one of the elements:

```
>>> fruit = ["banana", "apple", "quince"]
>>> fruit[0] = "pear"
>>> fruit[2] = "orange"
>>> fruit
['pear', 'apple', 'orange']
```

- The bracket operator applied to a list can appear anywhere in an expression.
- When it appears on the left side of an assignment, it changes one of the elements in the list, so the first element of fruit has been changed from "banana" to "pear", and the last from "quince" to "orange".
- An assignment to an element of a list is called item assignment.
- Item assignment does not work for strings:

```
>>> my_string = "TEST"
>>> my_string[2] = "X"
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

but it does for lists:

```
>>> my_list = ["T", "E", "S", "T"]
>>> my_list[2] = "X"
>>> my_list
['T', 'E', 'X', 'T']
```

With the slice operator we can update a whole sublist at once:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> a_list[1:3] = ["x", "y"]
>>> a_list
['a', 'x', 'y', 'd', 'e', 'f']
```

We can also remove elements from a list by assigning an empty list to them:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> a_list[1:3] = []
>>> a_list
['a', 'd', 'e', 'f']
```

- And we can add elements to a list by squeezing them into an empty slice at the desired location:

```
>>> a_list = ["a", "d", "f"]
>>> a_list[1:1] = ["b", "c"]
>>> a_list
['a', 'b', 'c', 'd', 'f']
>>> a_list[4:4] = ["e"]
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f']
```

List deletion

- Using slices to delete list elements can be error prone. Python provides an alternative that is more readable. The ***"del"*** statement removes an element from a list:

```
>>> a = ["one", "two", "three"]
>>> del a[1]
>>> a
['one', 'three']
```

- As you might expect, ***"del"*** causes a run time error if the index is out of range. You can also use del with a slice to delete a sub list:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> del a_list[1:5]
>>> a_list
['a', 'f']
```

- As usual, the sublist selected by slice contains all the elements up to, but not including, the second index.

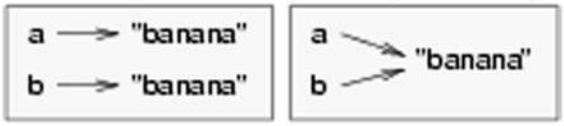
Objects and references

- After we execute these assignment statements

  a = "banana"

b = "banana"

- We know that **"a"** and **"b"** will refer to a string object with the letters "banana". But we don't know yet whether they point to the same string object.
- There are two possible ways the Python interpreter could arrange its memory:



- In one case, **"a"** and **"b"** refer to two different objects that have the same value. In the second case, they refer to the same object. We can test whether two names refer to the same object using the is operator:

>>> a is b

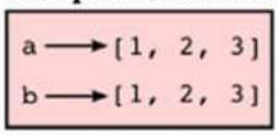
True

- This tells us that both **"a"** and **"b"** refer to the same object, and that it is the second of the two state snapshots that accurately describes the relationship. Since strings are immutable, Python optimizes resources by making two names that refer to the same string value refer to the same object.

- This is **not the case with lists**:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
>>> a is b
False
```

The state snapshot here looks like this:



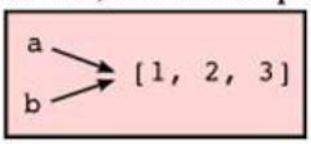
a and b have the same value but do not refer to the same object.

Aliasing

- Since variables refer to objects, if we assign one variable to another, both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True
```

In this case, the state snapshot looks like this:



- Because the same list has two different names, a and b, we say that it is aliased. Changes made with one alias affect the other:

>>> b[0] = 5

>>> a [5, 2, 3]

- Although this behavior can be useful, it is sometimes unexpected or undesirable. In general, it is safer to avoid aliasing when you are working with mutable objects. Of course, for immutable objects (i.e. strings, tuples), there's no problem - it is just not possible to change something and get a surprise when you access an alias name. That's why Python is free to alias strings (and any other immutable kinds of data) when it sees an opportunity to economize.

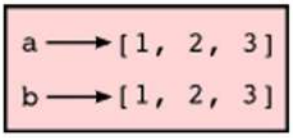
Cloning lists

- If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference.

- This process is sometimes called cloning, to avoid the ambiguity of the word copy. The easiest way to clone a list is to use the slice operator:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
```

- Taking any slice of a creates a new list. In this case the slice happens to consist of the whole list. So now the relationship is like this:



- Now we are free to make changes to *"b"* without worrying that we'll inadvertently be changing *"a"*:

```
>>> b[0] = 5
>>> a
[1, 2, 3]
```

Lists and "for" loops

The `for` loop also works with lists, as we've already seen. The generalized syntax of a `for` loop is:

```
for <VARIABLE> in <LIST>:
    <BODY>
```

So, as we've seen

```
friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
for friend in friends:
    print(friend)
```

It almost reads like English: For (every) friend in (the list of) friends, print (the name of the) friend. Any list expression can be used in a `for` loop:

```
for number in range(20):
    if number % 3 == 0:
        print(number)
for fruit in ["banana", "apple", "quince"]:
    print("I like to eat " + fruit + "s!")
```

- The first example prints all the multiples of 3 between 0 and 19. The second example expresses enthusiasm for various fruits. Since lists are mutable, we often want to traverse a list, changing each of its elements. The following squares all the numbers in the list *"xs"*:

```
xs = [1, 2, 3, 4, 5]
for i in range(len(xs)):
    xs[i] = xs[i]**2
```

- Take a moment to think about *range(len(xs))* until you understand how it works.
- In this example we are interested in both the value of an item, (we want to square that value), and its index (so that we can assign the new value to that position). This pattern is common enough that Python provides a nicer way to implement it:

```
xs = [1, 2, 3, 4, 5]
for (i, val) in enumerate(xs):
    xs[i] = val**2
```

- ***"enumerate"*** generates ***pairs of both (index, value) during the list traversal***. Try this next example to see more clearly how enumerate works:

```
for (i, v) in enumerate(["banana", "apple", "pear", "lemon"]):
    print(i, v)
```

```
0 banana
1 apple
2 pear
3 lemon
```

List parameters

- ***Passing a list as an argument actually passes a reference to the list, not a copy or clone of the list***. So parameter passing creates an alias for you: the caller has one variable referencing the list, and the called function has an alias, but there is only one underlying list object.
- For example, the function below takes a list as an argument and multiplies each element in the list by 2:

```
def double_stuff(stuff_list):
    for (index, stuff) in enumerate(stuff_list):
        stuff_list[index] = 2 * stuff
```

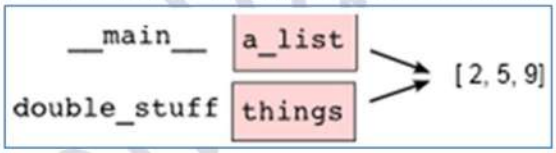
If we add the following onto our script:

```
things = [2, 5, 9]
double_stuff(things)
print(things)
```

When we run it we'll get:

```
[4, 10, 18]
```

- In the function above, the parameter ***stuff_list*** and the variable ***things*** are aliases for the same object. So before any changes to the elements in the list, the state snapshot looks like this:



- Since the list object is shared by two frames, we drew it between them.
- If a function modifies the items of a list parameter, the caller sees the change.

List methods

- The dot operator can also be used to access built-in methods of list objects. We'll start with the most useful method for adding something onto the end of an existing list:
- ***"append"*** is a list method which adds the argument passed to it to the end of the list. We'll use it heavily when we're creating new lists. Continuing with this example, we show several other list methods:

```
>>> mylist = []
>>> mylist.append(5)
>>> mylist.append(27)
>>> mylist.append(3)
>>> mylist.append(12)
>>> mylist
[5, 27, 3, 12]
```

```
>>> mylist.insert(1, 12)  # Insert 12 at pos 1, shift other items up
>>> mylist
[5, 12, 27, 3, 12]
>>> mylist.count(12)  # How many times is 12 in mylist?
2
>>> mylist.extend([5, 9, 5, 11]) # Put whole list onto end of mylist
>>> mylist
[5, 12, 27, 3, 12, 5, 9, 5, 11])
>>> mylist.index(9)  # Find index of first 9 in mylist
6
>>> mylist.reverse()
>>> mylist
[11, 5, 9, 5, 12, 3, 27, 12, 5]
>>> mylist.sort()
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 12, 27]
>>> mylist.remove(12) # Remove the first 12 in the list
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 27]
```

- Experiment and play with the list methods shown here and read their documentation until you feel confident that you understand how they work.

Pure functions and modifiers

- There is a difference between a pure function and one with side-effects. The difference is shown below as lists have some special gotcha's.
- Functions which take lists as arguments and change them during execution are called ***modifiers*** and the changes they make are called ***side effects.***
- A pure function does not produce side effects. It communicates with the calling program only through parameters, which it does not modify, and are turn value. Here is ***double_stuff written as a pure function***:

```
def double_stuff(a_list):
    """ Return a new list which contains
        doubles of the elements in a_list.
    """
    new_list = []
    for value in a_list:
        new_elem = 2 * value
        new_list.append(new_elem)

    return new_list
```

This version of `double_stuff` does not change its arguments:

```
>>> things = [2, 5, 9]
>>> more_things = double_stuff(things)
>>> things
[2, 5, 9]
>>> more_things
[4, 10, 18]
```

- An early rule we saw for assignment said ***"first evaluate the right hand side, then assign the resulting value to the variable".*** So it is quite safe to assign the function result to the same variable that was passed to the function:

```
>>>things=[2,5,9]
>>>things=double_stuff(things)
>>>things
[4,10,18]
```

Functions that produce lists

- The pure version of ***double_stuff*** above made use of an important pattern for your tool box. Whenever you need to write a function that creates and returns a list, the pattern is usually:

```
initialize a result variable to be an empty list
loop
   create a new element
   append it to result
return the result
```

- Let us show another use of this pattern. Assume you already have a function ***is_prime(x)*** that can test if ***"x"*** is prime. Write a function to return a list of all prime numbers less than ***"n"***:

```
def primes_lessthan(n):
    result = []
    for i in range(2, n):
        if is_prime(i):
            result.append(i)
    return result
```

Strings and lists

- ***Two of the most useful methods on strings involve conversion to and from lists of substrings.*** The split method breaks a string into a list of words***. By default, any number of whitespace characters is considered a word boundary***:

```
>>>song="The rain in Spain..."
>>>words = song.split()
>>>words
['The', 'rain', 'in', 'Spain...']
```

- An optional argument called a delimiter can be used to specify which string to use as the boundary marker between sub strings. The following example uses the string ***"ai"*** as the delimiter:

```
>>>song.split("ai")
['Ther', 'ninSp', 'n...']
```

- Notice that the delimiter doesn't appear in the result. The ***inverse of the "split"*** method is ***"join"***. You choose the desired separator string ***(often called the glue)*** and join the list with the glue between each of the elements:

```
>>>glue=";"
>>>phrase=glue.join(words)
>>>phrase
```

'The; rain; in; Spain...'

- The list that you glue together (words in this example) is not modified. Also, as these next examples show, you can use empty glue or multi-character strings as glue:

```
>>>"---".join(words)
'The---rain---in---Spain...'
>>>"".join(words)
'TheraininSpain...'
```

---

list and range

- Python has a ***built-in type conversion function*** called ***"list"*** that tries to turn whatever you give it into a list.

```
>>> letters = list("Crunchy Frog")
>>> letters
["C", "r", "u", "n", "c", "h", "y", " ", "F", "r", "o", "g"]
>>> "".join(letters)
'Crunchy Frog'
```

- ***One particular feature of range is that it doesn't instantly compute all its values: it "puts off" the computation, and does it on demand, or "lazily".***
- We'll say that it gives a promise to produce the values when they are needed. This is ***very convenient if your computation short-circuits a search and returns early, as in this case***:

```
def f(n):
 """ Find the first positive integer between 101 and less
    than n that is divisible by 21
 """
    for i in range(101, n):
       if (i % 21 == 0):
            return i

OUTPUT
print(f(110) == 105)
print(f(1000000000) == 105)
```

- In the second test, if range were to eagerly go about building a list with all those elements, you would soon exhaust your computer's available memory and crash the program.
- But it is cleverer than that! This computation works just fine, because the range object is just a promise to produce the elements if and when they are needed. Once the condition in the if becomes true, no further elements are generated, and the function returns.
- You'll sometimes find the lazy range wrapped in a call to list. This forces Python to turn the lazy promise into an actual list:

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Looping and lists

- Computers are useful because they can repeat computation, accurately and fast. So ***loops are going to be a central feature of almost all programs you encounter.***
- Lists are useful if you need to keep data for later computation. But if you don't need lists, it is probably better not to generate them.
- Here are two functions that both generate ten million random numbers and return the sum of the numbers. They both work.

```
import random
joe = random.Random()

def sum1():
    """ Build a list of random numbers, then sum them """
    xs = []
    for i in range(10000000):
        num = joe.randrange(1000)   # Generate one random number
        xs.append(num)              # Save it in our list
    tot = sum(xs)
    return tot

def sum2():
    """ Sum the random numbers as we generate them """
    tot = 0
    for i in range(10000000):
        num = joe.randrange(1000)
        tot += num
    return tot

print(sum1())
print(sum2())
```

- What reasons are there for preferring the second version here? (Hint: open a tool like the Performance Monitor on your computer and watch the memory usage.)
- In a similar way, when working with files, we often have an option to read the whole file contents into a single string, or we can read one line at a time and process each line as we read it.
- ***Line-at-a-time*** is the more traditional and perhaps safer way to do things—you'll be able to work comfortably no matter how large the file is. But you may find ***whole-file-at-once*** is sometimes more convenient!

Nested lists

- A nested list is a list that appears as an element in another list. In this list, the element with index 3 is a nested list:

  >>>nested=["hello",2.0,5,[10,20]]
- If we output the element at index 3, we get:

  >>>print(nested[3])

  [10,20]
- To extract an element from the nested list, we can proceed in two steps:

  >>>elem=nested[3]

  >>>elem[0]

  10
- Or we can combine them:

  >>> nested[3][1]

  20
- Bracket operators evaluate from left to right, so this expression gets the 3'th element of nested and extracts the 1'th element from it.

Matrices

- Nested lists are often used to represent matrices. For example, the matrix:
  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ might be represented as: ***>>> mx = [[1, 2, 3], [4, 5, 6], [7, 8, 9]].***
- ***"mx"*** is a list with three elements, where each element is a row of the matrix. We can select an entire row from the matrix in the usual way:

  >>> mx[1]
  [4, 5, 6]
- Or we can extract a single element from the matrix using the double-index form:

  >>> mx[1][2]
  6
- The first index selects the row, and the second index selects the column. Although this way of representing matrices is common, it is not the only possibility. A small variation is to use a list of columns instead of a list of rows

---

Glossary

aliases Multiple variables that contain references to the same object.

clone To create a new object that has the same value as an existing object. Copying a reference to an object creates an alias but doesn't clone the object.

delimiter A character or string used to indicate where a string should be split.

element One of the values in a list (or other sequence). The bracket operator selects elements of a list. Also called *item*.

immutable data value A data value which cannot be modified. Assignments to elements or slices (sub-parts) of immutable values cause a runtime error.

index An integer value that indicates the position of an item in a list. Indexes start from 0.

item See *element*.

list A collection of values, each in a fixed position within the list. Like other types `str`, `int`, `float`, etc. there is also a `list` type-converter function that tries to turn whatever argument you give it into a list.

list traversal The sequential accessing of each element in a list.

modifier A function which changes its arguments inside the function body. Only mutable types can be changed by modifiers.

mutable data value A data value which can be modified. The types of all mutable values are compound types. Lists and dictionaries are mutable; strings and tuples are not.

nested list A list that is an element of another list.

object A thing to which a variable can refer.

pattern A sequence of statements, or a style of coding something that has general applicability in a number of different situations. Part of becoming a mature Computer Scientist is to learn and establish the patterns and algorithms that form your toolkit. Patterns often correspond to your "mental chunking".

promise An object that promises to do some work or deliver some values if they're eventually needed, but it lazily puts off doing the work immediately. Calling `range` produces a promise.

pure function A function which has no side effects. Pure functions only make changes to the calling program through their return values.

sequence Any of the data types that consist of an ordered collection of elements, with each element identified by an index.

side effect A change in the state of a program made by calling a function. Side effects can only be produced by modifiers.

step size The interval between successive elements of a linear sequence. The third (and optional argument) to the `range` function is called the step size. If not specified, it defaults to 1.

5.4. Dictionaries (Pg 126– 129)

Dictionary operations, dictionary methods, aliasing and copying.

- All of the ***compound data types*** we have studied in detail so far — ***strings, lists, and tuples*** — are sequence types, which use integers as indices to access the values they contain within them.
- ***Dictionaries*** are yet ***another kind of compound type***. They are ***Python's built-in mapping type***. They map keys, which can be any immutable type, to values, which can be any type (heterogeneous), just like the elements of a list or tuple.
- As an example, we will ***create a dictionary*** to translate English words into Spanish. For this dictionary, the keys are strings. ***One way to create a dictionary*** is to ***start with the empty*** dictionary and add ***key:value*** pairs. The empty dictionary is ***denoted {}***:

```
>>> english_spanish = {}
>>> english_spanish["one"] = "uno"
>>> english_spanish["two"] = "dos"
>>> print(english_spanish)
{"two": "dos", "one": "uno"}
```

- The first assignment creates a dictionary named ***english_spanish;*** the other assignments add new ***key:value*** pairs to the dictionary. We can print the current value of the dictionary in the usual way.
- The ***key:value*** pairs of the dictionary are separated by commas. Each pair contains a key and a value separated by a colon.

Hashing

- The order of the pairs may not be what was expected. Python uses complex algorithms, designed for very fast access, to determine where the ***key:value*** pairs are stored in a dictionary. For our purposes we can think of this ordering as unpredictable. You also might wonder why we use dictionaries at all when the same concept of mapping a key to a value could be implemented using a list of tuples:

```
>>> {"apples": 430, "bananas": 312, "oranges": 525, "pears": 217}
{'pears': 217, 'apples': 430, 'oranges': 525, 'bananas': 312}
>>> [('apples', 430), ('bananas', 312), ('oranges', 525), ('pears', 217)]
[('apples', 430), ('bananas', 312), ('oranges', 525), ('pears', 217)]
```

- The reason is dictionaries are very fast, implemented using a technique called ***hashing***, which ***allows us to access a value very quickly***. By contrast, ***the list of tuples implementation is slow***. If we wanted to find a value associated with a key, we would have to iterate over every tuple, checking the 0th element. What if the key wasn't even in the list? We would have to get to the end of it to find out.
- Another way to create a dictionary is to provide a list of ***key:value*** pairs using the same syntax as the previous output:

```
>>> english_spanish = {"one": "uno", "two": "dos", "three": "tres"}
```

- It doesn't matter what order we write the pairs. The values in a dictionary are accessed with keys, not with indices, so there is no need to care about ordering.

- Here is how we use a key to look up the corresponding value:

```
>>> print(english_spanish["two"])
'dos'
```

- Lists, tuples, and strings have been called sequences, because their items occur in order. The dictionary is the first compound type that we've seen that is not a sequence, so we can't index or slice a dictionary.

Dictionary operations

- The ***"del"*** statement removes a ***key:value*** pair from a dictionary. For example, the following dictionary contains the names of various fruits and the number of each fruit in stock:

```
>>> inventory = {"apples": 430, "bananas": 312, "oranges": 525, "pears": 217}
>>> print(inventory) {'pears': 217, 'apples': 430, 'oranges': 525, 'bananas': 312}
```

- If someone buys all of the bananas, we can remove the entry from the dictionary:

```
>>> del inventory["bananas"]
>>> print(inventory)
{'apples': 430, 'oranges': 525, 'pears': 217}
```

- If we then try to see how many bananas we have, we get an error (because, yes, we have no bananas). (Try this!)
- Or if we're expecting more bananas soon, we might just change the value associated with bananas:

```
>>> inventory["bananas"] = 0
>>> print(inventory)
{'pears': 217, 'apples': 430, 'oranges': 525, 'bananas': 0}
```

- A new shipment of bananas arriving could be handled like this:

```
>>> inventory["bananas"] += 200
>>> print(inventory)
{'pears': 0, 'apples': 430, 'oranges': 525, 'bananas': 512}
```

- The ***"len"*** function also works on dictionaries; it returns the number of ***key:value*** pairs:

```
>>> len(inventory)
4
```

---

dictionary methods

- Dictionaries have a number of useful built-in methods.
- The ***keys*** method returns what Python 3 calls a view of its underlying ***keys***. A ***view*** object has some similarities to the ***range*** object we saw earlier — it is a lazy promise, to deliver its elements when they're needed by the rest of the program. We can iterate over the view, or turn the view into a list like this:

```
for key in english_spanish.keys(): # The order of the k's is not defined
        print("Got key", key, "which maps to value", english_spanish[key])
keys = list(english_spanish.keys())
print(keys)
```

OUTPUT

```
Got key three which maps to value tres
Got key two which maps to value dos
Got key one which maps to value uno
['three', 'two', 'one']
```

- It is so common to iterate over the ***keys*** in a dictionary that we can omit the keys method call in the ***for*** loop — iterating over a dictionary implicitly iterates over its keys:

```
for key in english_spanish:
    print("Got key", key)
```

- The ***values*** method is similar; it returns a view object which can be turned into a list:

```
>>> list(english_spanish.values())
['tres', 'dos', 'uno']
```

- The items method also returns a view, which promises a list of tuples — one tuple for each ***key:value*** pair:

```
>>> list(english_spanish.items())
[('three', 'tres'), ('two', 'dos'), ('one', 'uno')]
```

- Tuples are often useful for getting both the key and the value at the same time while we are looping:

Python Code	Output
`for (key,value) in english_spanish.items():` `  print("Got",key,"that maps to",value)`	*Got three that maps to tres* *Got two that maps to dos* *Got one that maps to uno*

• The ***in*** and ***not in*** operators can test if a key is in the dictionary:
`>>> "one" in english_spanish`
`>>> "six" in english_spanish`
`>>> "tres" in english_spanish`
# Note that 'in' tests keys, not values. 	
	True
	False
	False

- This method can be very useful, since looking up a non-existent key in a dictionary causes a runtime error:

```
>>> english_spanish["dog"]
Traceback (most recent call last):
  ...
KeyError: 'dog'
```

---

aliasing and copying

- Whenever two variables refer to the same object, changes to one affect the other.
- If we want to modify a dictionary and keep a copy of the original, use the copy method. For example, opposites is a dictionary that contains pairs of opposites:

```
>>> opposites = {"up": "down", "right": "wrong", "yes": "no"}
>>> alias = opposites
>>> copy = opposites.copy()  # Shallow copy
```

`alias` and `opposites` refer to the same object; `copy` refers to a fresh copy of the same dictionary. If we modify `alias`, `opposites` is also changed:

```
>>> alias["right"] = "left"
>>> opposites["right"]
'left'
```

If we modify `copy`, `opposites` is unchanged:

```
>>> copy["right"] = "privilege"
>>> opposites["right"]
'left'
```

END OF MODULE 2
