MODULE-2

STRINGS, TUPLES AND LISTS

SYLLABUS:
Strings: *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: *Tuples are used for grouping data, Tuple assignment, Tuples as return values, Composability of Data Structures.*
Lists: *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.*

CHAPTERS: *5.1, 5.2, 5.3*

TEXT BOOK: *Peter Wentworth, Jeffrey Elkner, Allen B. Downey and Chris Meyers- How to think like a computer scientist: learning with python 3. Green Tea Press, Wellesley, Massachusetts, 2020*

STRINGS - A COMPOUND DATA TYPE

- 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

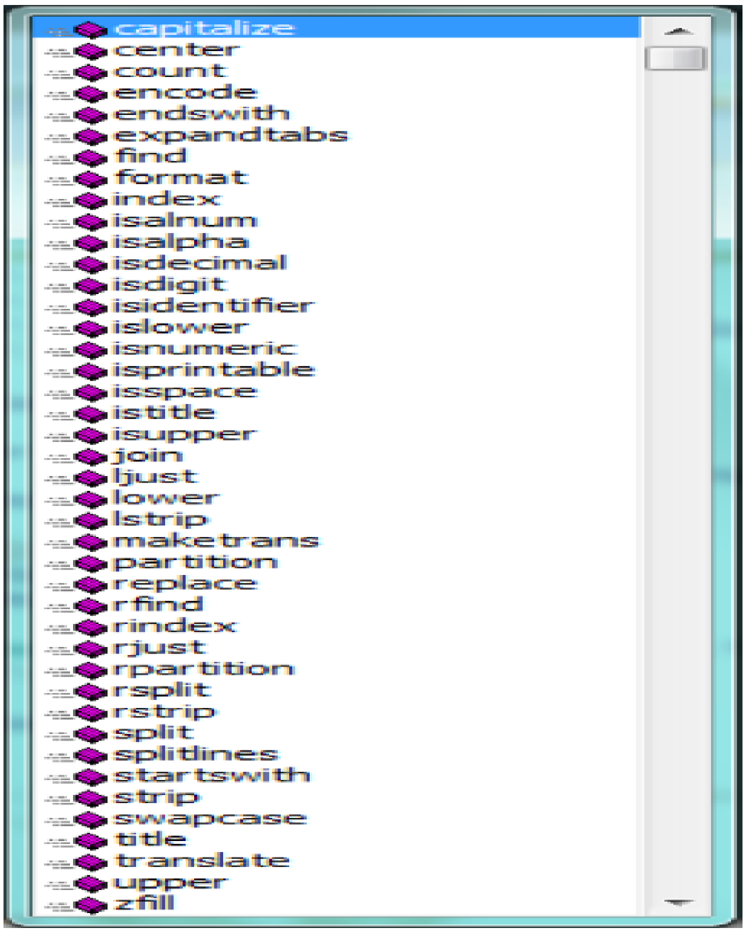
- Earlier in each turtle instance has its own attributes and a number of methods that can be applied to the instance. For example, we could set the turtle's color, and we wrote tess.turn(90).
- Just like a turtle, a string is also an object. So each string instance has its own attributes and methods. For example:

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

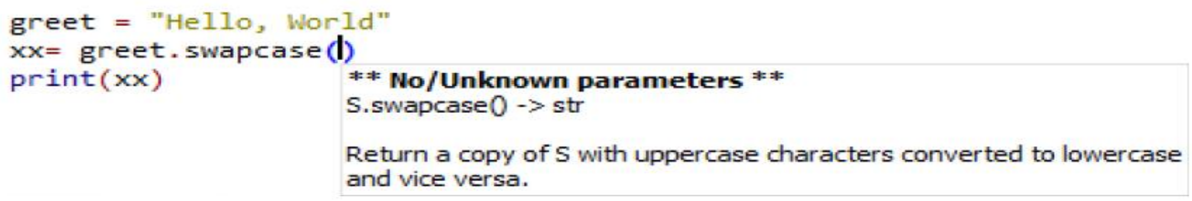
- upper is a method that can be invoked on any string object to create a new string, in which all the characters are in uppercase. (The original string our_string remains unchanged.) There are also methods such as lower, capitalize, and swapcase that do other interesting stuff.
- To learn what methods are available, can consult the Help documentation, look for string methods, and read the documentation. Or, if you're a bit lazier, simply 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 (there are around 70 of them — thank goodness we'll only use a few of those!) that could be used on your string.



- When typed 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).



```
greet = "Hello, World"
xx= greet.swapcase()
print(xx)
```

** No/Unknown parameters **
S.swapcase() -> str

Return a copy of S with uppercase characters converted to lowercase and vice versa.

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)
```

- 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:

```
a
```

- 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, we 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. And also 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:

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

- This 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.
- We've previously seen how the for loop can easily iterate over the elements in a list and it can do so for strings as well:

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

- Each time through the loop, the next character in the string is assigned to the variable c. 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 an abecedarian 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, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = "JKLMNOPQ"
suffix = "ack"
for p in prefixes:
print(p + suffix)
```

The output of this program is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

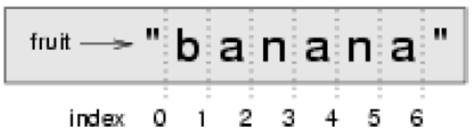
- Of course, that's not quite right because Ouack and Quack are misspelled.

SLICES

- A substring of a string is obtained by taking a slice. Similarly, we can slice a list to refer to some sublist 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", "Thandi", "Paris"]
>>> 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:



- 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: if you omit the first index (before the colon), the slice starts at the beginning of the string (or list). If you omit the second index, the slice extends to the end of the string (or list). 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'
```

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.

```
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.

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 we can't change an existing string. The best we 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 uppercase vowels would not be removed.

A FIND FUNCTION

- What does the following function do?

```
def my_find(haystack, needle):
    """
     Find and return the index of needle in haystack.
     Return -1 if needle does not occur in haystack.
    """
    for index, letter in enumerate(haystack):
        if letter == needle:
            return index
    return -1
```

- Compare the output of the code above with what Python does itself with the code below:

```
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)
```

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.

- Better still, 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

- 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

- 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.
- Consider 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 newlines 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
until they stop breathing completely. Once the heart stops the prey
is swallowed whole. The entire animal is digested in the snake's
stomach except for fur or feathers. What do you think happens to the fur,
feathers, beaks, and eggshells? The 'extra stuff' gets passed out as ---
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 Python 3 is to use the format method. 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 line 6 above!
- But there's more! Each of the replacement fields can also contain a format specification — it is always introduced by the : symbol (Line 13 above uses one.) This modifies how the substitutions are made into the template, and can control things like:
  - whether the field is aligned to the left <, center ^, or right >
  - the width allocated to the field within the result string (a number like 10)
  - the type of conversion (we'll initially only force conversion to float, f, as we did in line 13 of the code above, or perhaps we'll ask integer numbers to be converted to hexadecimal using x)
  - 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 needed 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("123456789 123456789 123456789 123456789 123456789 123456789")
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))
```

This script produces the output:

```
Pi to three decimal places is 3.142
123456789 123456789 123456789 123456789 123456789 123456789
|||Paris          |||     Whitney    |||         Hilton|||Born in 1981|||
The decimal value 123456 converts to hex value 1e240
```

➢ We 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!
   If you deposit $10 million into my bank account, I can
  double your money ...
"""
print(letter.format("Paris", "Whitney", "Hilton"))
print(letter.format("Bill", "Henry", "Gates"))
```

This produces the following:

```
Dear Paris Hilton.
 Paris, I have an interesting money-making proposition for you!
 If you deposit $10 million into my bank account, I can
 double your money ...


Dear Bill Gates.
 Bill, I have an interesting money-making proposition for you!
 If you deposit $10 million into my bank account I can
 double your money ...
```

➢ Also we get an index error if your placeholders 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:

```
print("i\ti**2\ti**3\ti**5\ti**10\ti**20")
for i in range(1, 11):
    print(i, "\t", i**2, "\t", i**3, "\t", i**5, "\t",
                                          i**10, "\t", i**20)
```

➢ This program prints out a table of various powers of the numbers from 1 to 10. (This assumes that the tab width is 8. You might see something even worse than this if you tab width is set to 4.) In its current form it relies on the tab character (\t) to align the columns of values, but this breaks down when the values in the table get larger than the tab width:

```
i       i**2    i**3    i**5    i**10   i**20
1       1       1       1       1       1
2       4       8       32      1024    1048576
3       9       27      243     59049   3486784401
4       16      64      1024    1048576         1099511627776
5       25      125     3125    9765625         95367431640625
6       36      216     7776    60466176        3656158440062976
7       49      343     16807   282475249       79792266297612001
8       64      512     32768   1073741824      1152921504606846976
9       81      729     59049   3486784401      12157665459056928801
10      100     1000    100000  10000000000     100000000000000000000
```

- 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. As you may have guessed by now, string formatting provides a much nicer solution. We can also right-justify each field:

```
layout = "{0:>4}{1:>6}{2:>6}{3:>8}{4:>13}{5:>24}"
print(layout.format("i", "i**2", "i**3", "i**5", "i**10", "i**20"))
for i in range(1, 11):
    print(layout.format(i, i**2, i**3, i**5, i**10, i**20))
```

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

```
   i  i**2  i**3    i**5        i**10                   i**20
   1     1     1       1            1                       1
   2     4     8      32         1024                 1048576
   3     9    27     243        59049              3486784401
   4    16    64    1024      1048576           1099511627776
   5    25   125    3125      9765625          95367431640625
   6    36   216    7776     60466176        3656158440062976
   7    49   343   16807    282475249       79792266297612001
   8    64   512   32768   1073741824     1152921504606846976
   9    81   729   59049   3486784401    12157665459056928801
  10   100  1000  100000  10000000000   100000000000000000000
```

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")
```

- The other thing that could be said somewhere around here, is that the parentheses are there to disambiguate. For example, if we have a tuple nested within another tuple and the parentheses weren't there, how would we tell where the nested tuple begins/ends? Also: the creation of an empty tuple is done like this: empty_tuple=()

- Tuples are useful for representing what other languages often call records (or structs) — some related information that belongs together, like your student record. There is no description of what each of these fields means, but we can guess. 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 immutable. 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 (but you're probably not likely to do that too often), 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):
    """ Return (circumference, area) of a circle of radius r """
    circumference = 2 * math.pi * r
    area = math.pi * r * r
    return (circumference, area)
```

COMPOSABILITY OF DATA STRUCTURES

- Earlier we could make a list of pairs, and we had 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.

LISTS

- A list is an ordered collection of values. The values that make up a list are called its elements, or its items. 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 can assign list values to variables or pass lists as parameters to functions:

```
>>> 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: [] (not to be confused with an empty list). 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])
```

- 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. The above sample doesn't need or use the index i 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

- The function len returns the length of a list, which is equal to the number of its elements. If 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, 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 sublists:

```
>>> 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 we might expect, del causes a runtime error if the index is out of range. We can use del with a slice to delete a sublist:

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

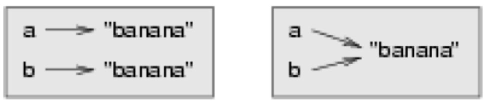
- As usual, the sublist selected by slice contains all the elements up to, but not including, the second index.

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:



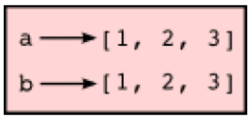
- In one case, a and b refer to two different objects that have the same value. In the second case, they refer to the same object. We can test whether two names refer to the same object using the is operator:

```
>>> 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.

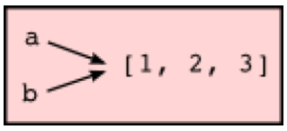


ALIASING

- 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]
```

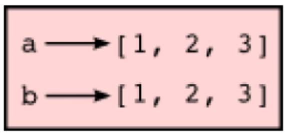
- Although this behavior can be useful, it is sometimes unexpected or undesirable. In general, it is safer to avoid aliasing when you are working with mutable objects.
- Of course, for immutable objects (i.e. strings, tuples), there's no problem, it is just not possible to change something and get a 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
```

- 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):
    """ Overwrite each element in a_list with double its value. """
    for (index, stuff) in enumerate(stuff_list):
        stuff_list[index] = 2 * stuff
```

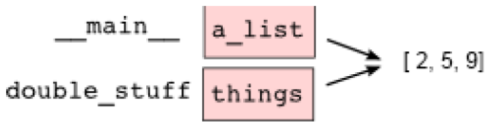
- If we add the following onto our script:

```
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:

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

- 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.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]
```

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 a return 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 toolbox. Whenever weneed 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):
    """ Return a list of all prime numbers less than 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 (which we’ve already seen) 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 substrings. The following example uses the string ai as the delimiter:

```
>>> song.split("ai")
['The r', 'n in Sp', 'n...']
```

- Notice that the delimiter doesn't appear in the result. The inverse of the split method is join. You choose a 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
print(f(110) == 105)
print(f(1000000000) == 105)
```

- 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.
- Find the lazy range wrapped in a call to list. This forces Python to turn the lazy promise into an actual list:

```
>>> range(10) # Create a lazy promise
range(0, 10)
>>> list(range(10)) # Call in the promise, to produce a list.
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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. Later we will see a more radical alternative using a dictionary.