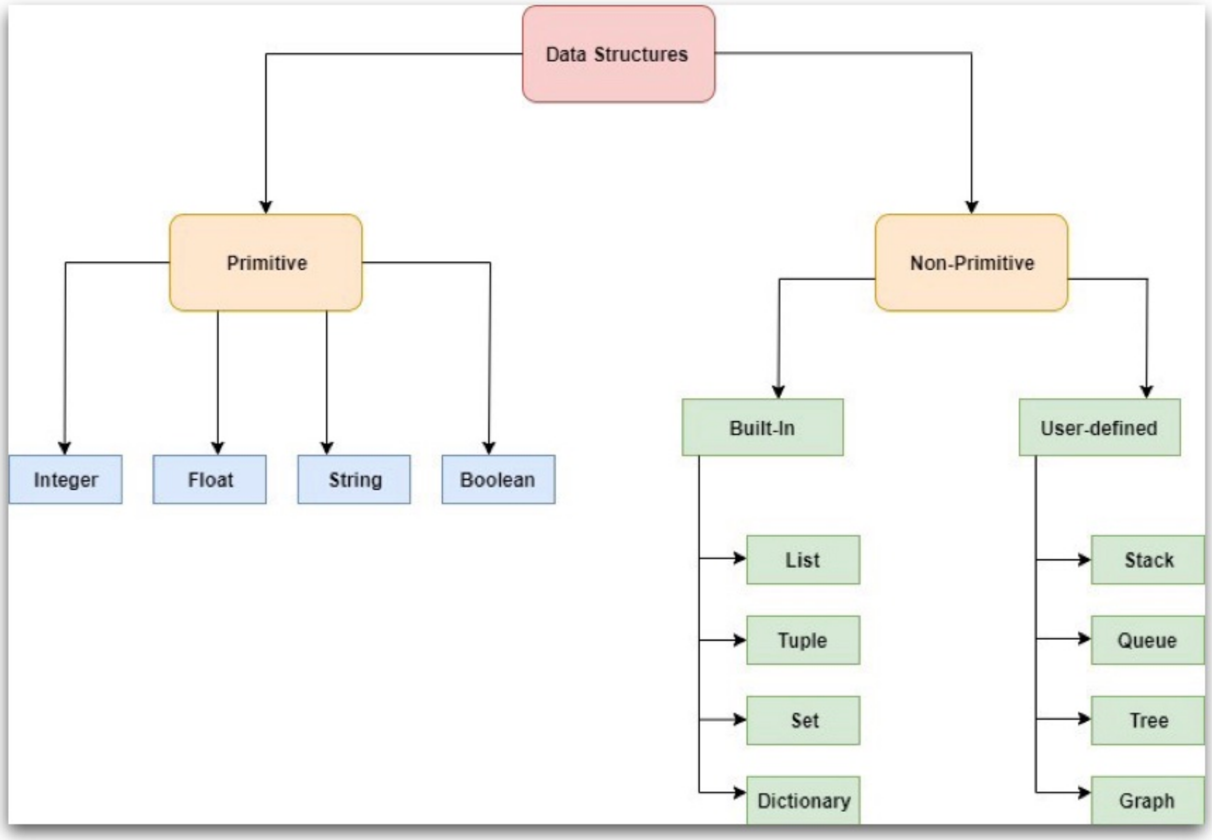
In [7]:
```python
from IPython import display
display.Image("images/chart-1.jpg")
```

Out[7]:



Week 3: Sequence Types & Dictionaries

Week 3. Version 1: Jan 16, 2022.

Discussion and demonstration.

This week's notebook introduces some vital data structures, mutability, and contains numerous optional examples. Focus on the **main themes** for the week, practicing your knowledge with the optional examples.

1. Sequences/Mutability
2. Bridge from last week:
   - Strings
   - Formatting
   - Range

3. Lists
    - Creating lists
    - in, not
    - sort versus sorted
    - min(), max()
    - sort demo with copied list
    - sort versus sorted
    - reversing lists
    - composite lists (lists nested in lists)
    - lists and mutability issues
    - copy versus deepcopy
    - pop()
    - recap of list methods
4. Activity
5. Optional/Study List Comphrensions (for week 4)
6. Tuples: often an alternative for permutations
7. Sets
8. Dictionaries: Often used with .json; handy for accessing data by key or value
9. Optional Exercises with .json, dictionaries, files

Sequences & Mutability

	sequenced	*non-sequenced*
mutable	list	dictionary
immutable	tuple	set

LIST: sequenced. This is the most versatile object; it's a set of any group of objects and mutable, meaning contents of a list can be changed, reordered, removed, new ones added, etc. [Hey, isn't that like an array? Yes, but with python we reserve "array" for the numpy library.]

TUPLE: sequenced, Like a list, but immutable.

SET: not a sequence, immutable. Common in GUI (as the x,y click location).

DICTIONARY: not a sequence, mutable; exceptionally useful because we can identify elements by value or name ("key").

Python relies on `{}` , `[]` , `()` instead of specific datatype call, as in Java or some other languages.

How might we use each of these? What are the benefits & liabilities?

Bridge from Week 2: Strings and Ranges

Last week we used Strings for splicing and accessing by index. All of the sequences use the same generic set of commands. Strings have a couple of obviously string-related behaviors like strip().

In [2]:
```
s = "Me thinks she doth protest too much."

print("Our demo string is [",s,"]")

print("\nHow many characters? ", len(s))

print("\nManipulate the output: \nUpper: ",s.upper(),
      "\nTitle:", s.title(),
      "\nlower:", s.lower(),
      "\nsplit:", s.split(),
      "\nfind 'doth':", s.find('doth'),
      "\nis 'yon' here?:", s.find('yon'),
      "\nHow many o's?:", s.count('o'))

st = s.split()[6]
print(st)

print("Let's remove the period:")
print(st.rstrip('.'))
```

```
Our demo string is [ Me thinks she doth protest too much. ]

How many characters?  36

Manipulate the output: 
Upper:  ME THINKS SHE DOTH PROTEST TOO MUCH. 
Title: Me Thinks She Doth Protest Too Much. 
lower: me thinks she doth protest too much. 
split: ['Me', 'thinks', 'she', 'doth', 'protest', 'too', 'much.'] 
find 'doth': 14 
is 'yon' here?: -1 
How many o's?: 4
much.
Let's remove the period:
much
```

Formatting note

Formatting in Python is on the move ...

In [3]:
```python
# old school
en = "What's up?"
zh = "你好"  # ni hao = you good
ru = "Какой язык!"

print("Old school with % substition using s for String")
print("Ni hao = %s" % zh)

# new school
print("\nNew school with format method")
print("Greetings = {}".format(en))

# newer school - fstring
print("\nNewer school with f string")
print(f"What a language = {ru}")

# template string (usually used for user-supplied data)
print("\n---- Template version ----")


from string import Template
t = Template("This is how to say hi: $hi")
# and then imagine later in code you tailor some message.
hi = "मज़ा लें!"
t.substitute(hi = hi)
```

```
Old school with % substition using s for String
Ni hao = 你好

New school with format method
Greetings = What's up?

Newer school with f string
What a language = Какой язык!

---- Template version ----
```

Out[3]:
```
'This is how to say hi: मज़ा लें!'
```

Range

Recall a range is a "sequence" and can use the usual `start : stop : step` idea as we did in string splicing (but using commas). It's useful, too, to cast the result of a range() into another object, say a `list`. Here are some examples.

In [4]:
```
a = range(10)
print(a)

b = range(0, 9)
print(b)

c = range(0,100,10)
print(c)

print("-"*50)
print("'a' is a: ", type(a))
aa = list(a)
print("'aa' is a: ", type(aa))

print("\nAnd casting c into a list we see ... ")
print(list(c))
```

```
range(0, 10)
range(0, 9)
range(0, 100, 10)
--------------------------------------------------
'a' is a:  <class 'range'>
'aa' is a:  <class 'list'>

And casting c into a list we see ...
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Lists, Tuples, Sets, & Dictionaries

Lists using []

Here are demos using lists, a mutable sequence.

Note: type, length, `in` , and `append()` .

In [5]:
```
# CREATE A LIST using []
listOfCafes = ['Café de Paris', 'Café des Fleurs', 'Napoléon', 'Maison Sauvag
print("Let's make sure of our data type: ", type(listOfCafes), " ID: ", id(li

x = listOfCafes[0]
print("\nReversed string using ::-1 = ", x[::-1])  # reverse - notice use sta

print("\n There are", len(listOfCafes), "cafés in the list.")
for i in listOfCafes:
    print("\t", i)

print("\nMy fav is ", listOfCafes[-1])
print("\nThe two favs are ", listOfCafes[2:4])

# I wanna add a new café if it isn't already there:
# and lets insert it at the front of the list, using 0

if "Café Aurore" not in listOfCafes:
    listOfCafes.insert(0, "Café Aurore")
print(listOfCafes)

# insert versus append
listOfRestaurants = ["Pré Catalan", "Rousseau", "Relais", "rue Rennes"]

print("\nHere is where I chow down in Paris")
print(listOfCafes + listOfRestaurants)


print(", ".join(listOfCafes))
print("C'est un miracle!")
```

```
Let's make sure of our data type:  <class 'list'>  ID:  140252680723520

Reversed string using ::-1 =  siraP ed éfaC

 There are 4 cafés in the list.
         Café de Paris
         Café des Fleurs
         Napoléon
         Maison Sauvage

My fav is  Maison Sauvage

The two favs are  ['Napoléon', 'Maison Sauvage']
['Café Aurore', 'Café de Paris', 'Café des Fleurs', 'Napoléon', 'Maison Sauva
ge']

Here is where I chow down in Paris
['Café Aurore', 'Café de Paris', 'Café des Fleurs', 'Napoléon', 'Maison Sauva
ge', 'Pré Catalan', 'Rousseau', 'Relais', 'rue Rennes']
Café Aurore, Café de Paris, Café des Fleurs, Napoléon, Maison Sauvage
C'est un miracle!
```

in and not

In our list let's check if something `is` in there or `not` ...

In [6]:
```
# IN example
if ("Napoléon" in listOfCafes):
    print("Vive l'Emperateur")
else:
    print("Yikes, vive la République")

print("\n","-"*30,"\nNote the use of not in:")

if ("Café Voltaire" not in listOfCafes):
    print("\tI bet Café Voltaire would be fun to visit.")
```

```
Vive l'Emperateur

 ------------------------------
Note the use of not in:
        I bet Café Voltaire would be fun to visit.
```

min, max

are useful not just for numbers but also the code for alphanumerics.

Consider how we can use min(), max() in many ways. Not just for values but maybe to ensure in our data cleaning that at least 1 example of some data is there.

min() and max() operate on numbers and on strings, using the utf-8 value: so "A" (65) will sort ahead of "a" (97) and so on.

In [7]:
```
years = [1888, 1823, 1723, 2002, 2021, 1066]
print("In the years list, the min and max are ", min(years), " and ", max(yea

people = ["Abraham", "करिशन", "Σοψρατεs", "BabyKitty"]
print("The min code points is", min(people), " and max by code points is ", m
```

```
In the years list, the min and max are  1066  and  2021

The min code points is Abraham  and max by code points is  करिशन
```

Sort Demo using a copied list.

The sort() command shows that the data in RAM aren't necessarily the same as the output. A list using `sort()` keeps original order and just changes the order in the output. Copying a list to another list creates two different items in the name space that point to the same object in object space.

The `sorted()` command sorts the list and stores the output in an entirely new object.

In [8]:
```
# sorting and ordering are important ...
print("id of the list of cafés: ",id(listOfCafes))

newList = sorted(listOfCafes)
print("\nJust made a new list, called 'newList'\n\t", newList)
print("which has id:", id(newList))

print("\nNow we have another list of just letters not in alphabetical order.
f = ['z','d','q']
print(f)
print(id(f))

f.sort()
print("Sorted version of the list", f)
print(id(f))
```

```
id of the list of cafés:  140252680723520

Just made a new list, called 'newList'
         ['Café Aurore', 'Café de Paris', 'Café des Fleurs', 'Maison Sauvage
', 'Napoléon']
which has id: 140252680721024

Now we have another list of just letters not in alphabetical order. Compare t
he outputs and their IDs.
['z', 'd', 'q']
140252681037568
Sorted version of the list ['d', 'q', 'z']
140252681037568
```

In [9]:
```
l1 = ['g', 'h', 'i', 'z', 'b', 'a']
print("l1 = ", l1)
print("first element of l1 = ", l1[0])
print("id: ", id(l1), "\n", "-"*20)

# COPY l1 to l2
print("Let's copy: make a new list (l2) from old list (l1)")
l2 = l1
print(l2[0])

print("")
print("IDs of both lists: 1 ", id(l1), " 2", id(l2))

# are l1 and l2 the same?
print("Let's change some values in one list and see the changes.")
print("l1[0] will be set to the number 8.  What happens to l2?")
# update values
l1[0] = '8'
print("l1: ",l1)
print("l2: ",l2)

print("-"*50)
print("Now lets create a _NEW_ object using the 'sorted()' command.")
# COPY TO A NEW OBJECT
l3 = sorted(l1)
print("this is l3: ", l3)

print("\n")
print("let's compare the IDs of all 3 lists:")
print("\tl1: ",id(l1), "\n\tl2: ",id(l2), "\n\tl3: ",id(l3))

# Changing a value in l3 (the sorted copy)
l3[0] = 'Q'

print("\ntest:\n1 ",l1, "\n2 ", l2, "\n3 ", l3)
```

```
l1 =  ['g', 'h', 'i', 'z', 'b', 'a']
first element of l1 =  g
id:  140252679829440
 --------------------
Let's copy: make a new list (l2) from old list (l1)
g

IDs of both lists: 1  140252679829440  2 140252679829440
Let's change some values in one list and see the changes.
l1[0] will be set to the number 8.  What happens to l2?
l1:  ['8', 'h', 'i', 'z', 'b', 'a']
l2:  ['8', 'h', 'i', 'z', 'b', 'a']
--------------------------------------------------
Now lets create a _NEW_ object using the 'sorted()' command.
this is l3:  ['8', 'a', 'b', 'h', 'i', 'z']


let's compare the IDs of all 3 lists:
	l1:  140252679829440
	l2:  140252679829440
	l3:  140252681493312

test:
```

```
1  ['8', 'h', 'i', 'z', 'b', 'a']
2  ['8', 'h', 'i', 'z', 'b', 'a']
3  ['Q', 'a', 'b', 'h', 'i', 'z']
```

Reverse Demo

In [10]:

```
demolist = ["我", "b", "z", "f", "j", "क", "é", "ñ"]
print("Original list:\n\t", demolist)
print("ID: ", id(demolist))

demolist.sort()
print("\nSorted list: ", demolist)
print("ID: ", id(demolist))

# the output is based on the characters Unicode code point value.
# we'll look at this later in the course.

print("\nLet's reverse the order using reverse = True ... ")
demolist.sort(reverse = True)
print("\t", demolist)

# another python shortcut ... try .reverse()
demolist.reverse()
print("\nusing '.reverse()'")
print("\t", demolist)
```

```
Original list:
         ['我', 'b', 'z', 'f', 'j', 'क', 'é', 'ñ']
ID:  140252680651968

Sorted list:  ['b', 'f', 'j', 'z', 'é', 'ñ', 'क', '我']
ID:  140252680651968

Let's reverse the order using reverse = True ...
         ['我', 'क', 'ñ', 'é', 'z', 'j', 'f', 'b']

using '.reverse()'
         ['b', 'f', 'j', 'z', 'é', 'ñ', 'क', '我']
```

Lists INSIDE lists: Composites

A very important and common event is to import data where there's a list and inside that list is a set of other list data. Practice identifying what looks like a "multidimensional" list is important. [BTW, some dictionaries often contain subsets as lists, too.]

Notice the `[ ]` used inside of the outer [ ].

In [11]:
```
list_names = ['Caleb', 'Gerald', 'Elizabeth', 'Ming', 'Sri']
list_skills = ['programming', 'ux', 'analysis', 'comm', 'design']

list_names.append(list_skills)
print(list_names)

print("-"*50)
# using index numbers to retrieve values
print("list_skills[0] =", list_skills[0])
print("list_names[0]  = ",list_names[0])

print("-"*50)
# now we'll use multidimensional address
print("the 5th element of the appended list is the nested list of activities.
print("list_names[5]=", list_names[5])

print("list_names[5][1]=", list_names[5][1])
```

```
['Caleb', 'Gerald', 'Elizabeth', 'Ming', 'Sri', ['programming', 'ux', 'analys
is', 'comm', 'design']]
--------------------------------------------------
list_skills[0] = programming
list_names[0]  =  Caleb
--------------------------------------------------
the 5th element of the appended list is the nested list of activities.
So [5][1] means 'go to the 5th element (oh, a list!) and then progress in tha
t list to the 1st element.'
list_names[5]= ['programming', 'ux', 'analysis', 'comm', 'design']
list_names[5][1]= ux
```

Copy versus Deepcopy

Copy creates a supposedly superficial level of copying data - but its really copies the references to the objects in the original list.

Supposedly if I change a value in the SOURCE it should be reflected, too, in the copy.

In [12]:
```
x = [ ["a", "b", "c"], ["X", "Y", "Z"] ]

x2 = list(x)
print(x)
print(x2)
print("\n")

# extend the first list and see if they're different elements
x.append(["Cat", "Dog"])
print("x  = ", x)
print("x2 = ", x2)

# so I can extend the source and not bother the copy.
# what if we change a value in the original?
x[0][0] = "SUKY"
print("\n", x)
print(x2)

# so, notice that we can append to the source and it won't
# appear in the copy.  BUT if we change a value in the
# original, it will be changed in the copy.
# this is a SHALLOW copy because in the end both have
# different names in name space but the same references
# in object space.
```

```
[['a', 'b', 'c'], ['X', 'Y', 'Z']]
[['a', 'b', 'c'], ['X', 'Y', 'Z']]


x  =  [['a', 'b', 'c'], ['X', 'Y', 'Z'], ['Cat', 'Dog']]
x2 =  [['a', 'b', 'c'], ['X', 'Y', 'Z']]

 [['SUKY', 'b', 'c'], ['X', 'Y', 'Z'], ['Cat', 'Dog']]
[['SUKY', 'b', 'c'], ['X', 'Y', 'Z']]
```

In [13]:
```
""" DEEP COPY """
import copy

y = [['a', b', c'], ['d', 'e', 'f']]
y2 = copy.deepcopy(y)

print("y  = ", y)
print("y2 = ", y2)

# the original and deepcopy are supposed to be
# unrelated objects WITH THEIR OWN copies -
# they don't share references to the data.
print('\n')
y[0][0] = 'FISH'
print("y with new value = ", y)
print("y2 copy values   = ", y2)
```

```
y  =  [['a', b', c'], ['d', 'e', 'f']]
y2 =  [['a', b', c'], ['d', 'e', 'f']]


y with new value =  [['FISH', b', c'], ['d', 'e', 'f']]
y2 copy values   =  [['a', b', c'], ['d', 'e', 'f']]
```

Pop!

is a common way to go to the end of the list and "pop" one off ... and automatically move to the left for the new end of the list 'til we run out of items in the list.

In [14]:
```
list4 = [1, 2, 3, 4, 5, 6, 7, 8, 9]

print(list4)
print(list4.pop())
print(list4) # notice that pop() removes the popped item from list

print(list4.pop(3))
print(list4)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
9
[1, 2, 3, 4, 5, 6, 7, 8]
4
[1, 2, 3, 5, 6, 7, 8]
```

Lists and Mutability Issues:

notice two lists, one with append() and one with extend()

In [15]:
```
print("_"*30, "\nUsing append()")
list_a = ["Newport", "Monterey", "Antibes"]
list_b = [2500, 4000, 3500]

print(list_a, "id: ", id(list_a))
print(list_b, "id: ", id(list_b))

print("Appending... ")
list_a.append(list_b)
print(list_a, "\nid: ", id(list_a))
print("Result is a list-in-a-list.")

print("_"*30, "\nUsing extend()")
list_c = ["Paris", "Strasbourg", "München"]
list_d = ["Roma", "Ticino", "Capri"]

print(list_c, "id: ", id(list_c), "\n", list_d, "id: ", id(list_d))
list_c.extend(list_d)
print("list_c has been extended by list_d - note the output\n", list_c)
print(id(list_c))
```

```
______________________________
Using append()
['Newport', 'Monterey', 'Antibes'] id:  140252680549120
[2500, 4000, 3500] id:  140252680549056
Appending...
['Newport', 'Monterey', 'Antibes', [2500, 4000, 3500]]
id:  140252680549120
```

```
Result is a list-in-a-list.
______________________________
Using extend()
['Paris', 'Strasbourg', 'München'] id:  140252680551360
 ['Roma', 'Ticino', 'Capri'] id:  140252680549184
list_c has been extended by list_d - note the output
 ['Paris', 'Strasbourg', 'München', 'Roma', 'Ticino', 'Capri']
```

Recap of List Methods

- `myList.insert(index, value)`
- `myList.pop(x)` : pops the last value by default. Takes also argument for particular index.
- `myList.remove()` : removes the first instance of a value
- `myList.sort()` : sorts the list and outputs in order
- `sorted(myList)` : returns a new list, in order
- `myList.reverse()` : Outputs the list in reverse.
- `myList.append("x")` : Adds the "x" to the end of the list
- `myList.extend(otherList)` : adds item from otherList to the end of myList
- `myList[n] = 'z'` : swaps out item at index[*n*] for whatever z stands for
- `myList.clear()` : keeps the variable named myList and removes all its members
- `del(myList[n])` : delete item at index *n* from List
- `copy()`
- `copy.deepcopy(otherlist)`

Activity

Recall and practice `range(start, stop(exclusive), step)`

Practice making these outputs:

- `[1, 2, 3, 4, 5, 6, 7, 8, 9]`
- `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
- `[2, 4, 6, 8, 10, 12]`
- `[2, 4, 6, 8, 10, 12, 13, 14, 15, 17, 19]`
- `[-1, 0, 1, 2, 3]`
- `[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]`

> ## Optional: List Comprehensions

> In this section we introduce and sequentially build on the idea of a list comprehensions.
>
> In the first example we use a for...if loop idea for a couple of lines of code. And then do the same in a *single* line.

In [16]:
```
alphabet = ["a", "b", "c", "d", "z"]
new_alphabet = []  # empty list

for i in alphabet:
    if "c" in i:
        new_alphabet.append(i)

print(new_alphabet)
print("\nAnd now list comprehension")

# and now the single-line version, a list comprehension
# new_data = [temp_var FOR temp_var in LIST IF some condition is met]
new_alphabet2 = [i for i in alphabet if "c" in i]
print(new_alphabet2)

# NOTE! Last week we learned that python doesn't use ! but - it does!
# here we see is applied in a pretty traditional way ...
not_alphabet = [i for i in alphabet if i != "c"]

# why not use "not"?  Because how would we construct the
# syntax?  Compare this notion (it's wrong, btw)
# not_alphabet2 = [i for i in alphabet if i not == "c"]
```

```
['c']

And now list comprehension
['c']
```

And a python shortcut ... with no `if` statement.

In [17]:
```
new_alphabet3 = [i for i in alphabet]
print(new_alphabet3)
```

```
['a', 'b', 'c', 'd', 'z']
```

Summary

`newlist = [expression for item in iterable if condition == True]`

Iterable

An *iterable* can be any iterable object, like a list or a tuple, or set ...

Here we use the oh-so-popular `range()` function to create an iterable and then create more iterables with conditions.

In [18]:
```
print(range(5))

list1 = [x for x in range(5)]
print(list1)

# now with a condition
# instead of writing an if code block ...
list2 = [x for x in range(5) if x < 3]
print(list2)
```

```
range(0, 5)
[0, 1, 2, 3, 4]
[0, 1, 2]
```

Example What if you want you to EXTRACT a subset of data? If you have an iterable and you check each element from that iterable, if the iterable isn't something you want, we can skip it and keep going.

Our original list contains a bunch of letters. We want to create a new list ONLY from letters that are not "g".

In [19]:
```
vowels = ['a','e','i','o','u']
alphabet = ['a','b','c','d','e','f','g','h','i','j','k']

# the for ... if way
newlist = []
for x in alphabet:
    if x == "g":
        newlist.append(x)

# the list comprehension way
newlist2 = [x for x in alphabet if x != "g"]
print(newlist2)

# in this example we add "if x not in vowels" -
# so x taken from alphabet is compared to members in
# the vowels list ... if the "x" is not in vowels,
# save the element (x's value) to our new list.
print("")
newlist3 = [x for x in alphabet if x not in vowels]
print("No vowels: ", newlist3)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'h', 'i', 'j', 'k']

No vowels:  ['b', 'c', 'd', 'f', 'g', 'h', 'j', 'k']
```

> ## End of the optional list materials.

Tuples

Sequential demos building up from creating a tuple with () and then emulating using data for a report.

Tuples are used to store multiple items in a single variable (as are the others we've seen); but a tuple is a collection which is

- ordered
- unchangeable
- allows duplicate data.

We use ( ) to create a tuple.

In [20]:
```
# create a tuple
clicked = (5, 3)

print(clicked[0])

t = ("Ryan", 30, "Trainer")
print(t)
print(t[0])

print("\n")
print("notice that we cannot append data nor cannot change the value of an ex
print("Appending... ")
t.append('Boston')
```

```
5
('Ryan', 30, 'Trainer')
Ryan


notice that we cannot append data nor cannot change the value of an existing
tuple element.
Appending...
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [20], in <module>
     11 print("notice that we cannot append data nor cannot change the value
of an existing tuple element.")
     12 print("Appending... ")
---> 13 t.append('Boston')

AttributeError: 'tuple' object has no attribute 'append'
```

In [21]:
```
# note, tho, if the tuple contains a mutable object, like a list, we can upda
a = ([1, 2, 3], 'a', 'b')
print(a)

print("After ... ")
a[0].append(4)
print(a)
```

```
([1, 2, 3], 'a', 'b')
After ...
([1, 2, 3, 4], 'a', 'b')
```

Packing a tuple

... means assigning a name to data in a tuple. Notice the left/right syntax. Left (packing) creates the tuple; creating a "right tuple" assigns names to our tuple data.

In [22]:
```
""" access the data in the tuple.
    first create the tuple and use an index number to get value
"""
# tuple packing [left] and tuple assignment [right]
students = ("Tom", "Maria", "Zeus")
print(students[2])

""" use right tuple assigment to create names for our values """
(student1, student2, student3) = students

""" now we can use the unique name for returning values """
print(student1)

# linking two tuples ...
scores = (50, 40, 52)
print(students + scores)
print(students[0], " received ", scores[0] )
print(student2, " received ", scores[1] )
```

```
Zeus
Tom
('Tom', 'Maria', 'Zeus', 50, 40, 52)
Tom  received  50
Maria  received  40
```

Even tho tuples are immutable, their elements are indexed so we can have multiple elements with the same *values*. `t = ("a", "b", "c", "a")` is okay.

In [23]:
```
t = ("a", "b", "c", "a")
print(t[0])

print(t[:-1]) # return a, b, c
print(t[:])   # just like other sequences : = every element
print(t[0:2])

if "c" in t:
    print("yup, all these sequences work alike.")
```

```
a
('a', 'b', 'c')
('a', 'b', 'c', 'a')
('a', 'b')
yup, all these sequences work alike.
```

But what if I need to add items to a tuple? One way is to conver the tuple to a list, add the items, and then conver the list back to a tuple.

Looping thru tuple using `while`, `for`, and `range()`

In [24]:
```python
# using a for statement
for i in range(len(t)):
    print(t[i])

print("-"*40)

# using a while statement
i = 0
lengthOfTuple = len(t)

while i < lengthOfTuple:
    print(t[i])
    i += 1

print("_"*50, "\nIterate thru tuple using range():")
mystudents = ("Tom", "José", "Fifi")
for i in range(len(mystudents)):
    print("i=",i, "\t", mystudents[i])
```

```
a
b
c
a
----------------------------------------
a
b
c
a
__________________________________________________
Iterate thru tuple using range():
i= 0     Tom
i= 1     José
i= 2     Fifi
```

A note about append() with tuple

We can't change the value of an existing tuple element. BUT we can append() a value to an existing tuple element. (Guess why.)

In [25]:
```python
a = ([1, 2, 3,], 2, 3)
print(a)

a[0].append(5)
print(a)
```

```
([1, 2, 3], 2, 3)
([1, 2, 3, 5], 2, 3)
```

> Optional Example:

Accessing data in nested tuples

> imaging we have been given data as a tuple and must make a report...

In [3]:
```
studentRecords = (
    ("Lars", "Ingersol"),
    ("Enrolled", 2022),
    "Classes", ("Chemistry", "BioChem"),
                     [ ("Intro", 99),
                       ("Organic", 99),
                       ("Inorganic", 30),
                       ("Biochem", 98),
                       ("Biochem 2", 98),
                       ("Quantum Chem", 100) ])

print("Last name: ", studentRecords[0][1])
print("Showing: ", studentRecords[2])
print("Majors: ", studentRecords[3])
print(studentRecords[4])

print(studentRecords[4][1])

print("_"*30)
x = 0
cnt = 0
for i in studentRecords[4]:
    print("Grades: ", i[1])
    x += i[1]
    cnt += 1


avg = x/cnt
print("Average ", avg)

# yikes, can we round?
print("\n\tRound:", round(x/cnt))

print("\nSeveral Formatting Style:")
print("\tUsing % style for floats:", "%.2f" % avg)

print("\tUse round() as part of formatting: ", "%.2f" % round(avg, 2))

print("\tUsing colon version: ", "{:.2f}".format(avg))

print("\t{:.2f}".format(round(avg, 2)))

# see also https://docs.python.org/3/library/string.html#format-specification
print("-"*30)
print("\nNote the combination of [] and splice commands.")
print(studentRecords[0][1][:])
```

```
Last name:  Ingersol
Showing:  Classes
Majors:  ('Chemistry', 'BioChem')
[('Intro', 99), ('Organic', 99), ('Inorganic', 30), ('Biochem', 98), ('Bioche
m 2', 98), ('Quantum Chem', 100)]
('Organic', 99)
```

```
______________________________
Grades:  99
Grades:  99
Grades:  30
Grades:  98
Grades:  98
Grades:  100
Average  87.33333333333333

        Round: 87

Several Formatting Style:
        Using % style for floats: 87.33
        Use round() as part of formatting:  87.33
        Using colon version:  87.33
        87.33
------------------------------

Note the combination of [] and splice commands.
Ingersol
```

Sets

- cannot have duplicate datas, because it's their *values* that are considered, unlike tuples that use a unique index
- are immutible

Sets are reminiscent of SQL and Venn diagrams in terms of set membership, too.

Use the `{ }` to create a set.

In [27]:
```
# using {} to create a set
mycats = {"Suky", "BabyKitty", "Bix"}

answers = {True, False, True, True}
print(mycats)
print(answers)

# see, no dupes in a set.  They get ignored.
```

```
{'Suky', 'BabyKitty', 'Bix'}
{False, True}
```

In [28]:
```
# poll any of these containers to see if there's an element
# and get a value ...
print( "Bix" in mycats )

#but ...
mycats.add("Kiki")
print(mycats)

tempcats = sorted(mycats) # outputs the sorted data as a list
print(tempcats)
print(type(tempcats))
```

```
True
{'Suky', 'Kiki', 'BabyKitty', 'Bix'}
['BabyKitty', 'Bix', 'Kiki', 'Suky']
<class 'list'>
```

In [29]:
```
# examples
mydogs = set(["Rex", "Koshka", "Mozart"])
print(mydogs)

for dog in mydogs:
    print(dog, end="")  # output surprising the formatting

print("\n","-"*40,"\nCan return true/false if in the set")
print("Rex" in mydogs)

print("\n","-"*40,"\nCan add() data, too!")
mydogs.add("Clive")
print(mydogs)
```

```
{'Koshka', 'Mozart', 'Rex'}
KoshkaMozartRex
 ----------------------------------------
Can return true/false if in the set
True

 ----------------------------------------
Can add() data, too!
{'Clive', 'Koshka', 'Mozart', 'Rex'}
```

Recap of Sets

- Cannot have duplicate data
- `add()`
- `remove()`
- `clear()`
- `copy()` (a "shallow" copy")
- `pop()`
- `update()`
- `union()`
- `difference()`
- `intersection()`

- `issuperset()`
- `issubset()`
- `isdisjoint()`

Dictionary

- mutible
- non-sequenced
- cannot have duplicate keys

For lots of examples using dictionaries and .json files demonstrated below.

Dictionaries are very useful! and can be instantiated in many ways. Note that we can use a "dictionary literal", or as a "list of tuples". They consist of `key` and `value` .

We can get the value by using a key name:

```
dict1['fred'] would return a value associated with that key.
```

A dictionary is created by a syntax similar to sets:

> myDict = {"Kim": 3, "Chris": 4, "Anthony": 2, "Avik", 2}

Note the `key:value` combination.
We can access data by its key, or by its value; or by all keys and by all values.

In this example two authors are identified (“OBrien” and “Kipling”;) they serve as the “key”.

Their works *Master & Commander* and *Gunda Din* are the “values”. We cannot use indexing (hence the error). But we can use the key to find the value.

Remember that dictionaries cannot have duplicate keys.

In [4]:
```
d1 = {'toby': 1, 'brian': 4, 'dave': 2, 'swampna': 5} # dict literal
d2 = dict([ ('lars', 1), ('rob', 4), ('roberto', 2), ('haerang', 5)])
# list of tuples

dflowers = {'azelia': 1, 'roses': 4, 'mum': 2}
d2 = dict([ ('lars', 1), ('rob', 4), ('roberto', 2), ('haerang', 5)])


del(d2['rob'])
# d2.pop('key', 'default_value')
# d2.get('key', 'default_value')

print("\nGetting the value from a tuple associated with 'lars' in a dictionar

d1.clear()
d2.update(dflowers)     # append a second dictionary

print('\nkeys: ', "-"*50)
print(d2.keys())

print('\nvalues: ', "-"*50)
print(d2.values())

print('\nitems: ', "-"*50)
print(d2.items())

print("\nprinting elements in d2: ", d2)
```

```
Getting the value from a tuple associated with 'lars' in a dictionary:  1

keys:  --------------------------------------------------
dict_keys(['lars', 'roberto', 'haerang', 'azelia', 'roses', 'mum'])

values:  --------------------------------------------------
dict_values([1, 2, 5, 1, 4, 2])

items:  --------------------------------------------------
dict_items([('lars', 1), ('roberto', 2), ('haerang', 5), ('azelia', 1), ('ros
es', 4), ('mum', 2)])

printing elements in d2:  {'lars': 1, 'roberto': 2, 'haerang': 5, 'azelia':
1, 'roses': 4, 'mum': 2}
```

Practice creating dictionaries and extracting data by the key or by the value.

In [10]:
```
from IPython import display
display.Image("images/t-shirt.jpg")
```

```
Out[10]:
```

Dictionaries and .JSON

Just a note in passing ... JavaScript Object Notation (.json) files are used a lot to share data between systems. Notice that GitHub itself uses .json (if you view the raw mode). Important to note the syntax and { } for blocks in .json - the file won't work if not perfect.

- The format of a data file dictates how we can integrate the contents into our programs.
- Least structured is a “flat-file” - a stream of letters/numbers.
- More structured is a flat-file that relies on its tabs, new-lines for structure (common in bioinformatics)
- Next are files with a delimiter - meaning a special character to indicate the

> end of block of data. Most often a tab (\t).
> - From here we have .xml and .json files, which structures can be read into our code via a dictionary.
> - (Optional tech note: .xml files are often converted into classes + methods and use a “factory” class to create an hierarchy of nodes.)

End of the main points.

Optional Examples / Study section

> ## Optional Example: List & Dictionary Activity
>
> We are now going to solve a very popular problem: How do you count the words in a document? While the solution here is simple, you will see in later courses that this is an excellent first problem when learning how to massively parallelize your code across a cluster of computers.
> The activity will guide you to the solution in a series of steps.

In [32]:

```
# Open the file in read mode
text = open("sample.txt", "r")

# Create an empty dictionary
d = dict()

# Loop through each line of the file
for line in text:
    # Remove the leading spaces and newline character
    line = line.strip()

    # Convert the characters in line to
    # lowercase to avoid case mismatch
    line = line.lower()

    # Split the line into words # PAY ATTENTION TO DELIMITERS
    words = line.split(" ")

    # Iterate over each word in line # USUALLY SINGULAR FOR NODE; PLURAL FOR
    for word in words:
        # Check if the word is already in dictionary
        if word in d:
            # Increment count of word by 1
            d[word] = d[word] + 1
        else:
            # Add the word to dictionary with count 1
            d[word] = 1

# Print the contents of dictionary
for key in list(d.keys()):
    print(key, ":", d[key])
```

```
balzac's : 3
extensive : 1
use : 2
of : 17
detail, : 1
especially : 1
the : 18
detail : 1
objects, : 1
to : 6
illustrate : 1
lives : 1
his : 3
characters : 1
made : 1
him : 1
an : 1
early : 1
pioneer : 1
literary : 1
realism. : 1
while : 1
he : 2
admired : 1
and : 3
drew : 1
inspiration : 1
from : 1
romantic : 1
style : 1
scottish : 1
novelist : 2
walter : 1
scott, : 1
balzac : 4
sought : 1
depict : 1
human : 2
existence : 1
through : 3
particulars. : 1
in : 3
preface : 1
first : 1
edition : 1
scènes : 1
de : 2
la : 1
vie : 1
privée, : 1
wrote: : 1
"the : 1
author : 1
firmly : 1
believes : 1
that : 2
details : 1
alone : 1
will : 1
henceforth : 1
```

```
determine : 1
merit : 1
works". : 1
plentiful : 1
descriptions : 2
décor, : 1
clothing, : 1
possessions : 1
help : 1
breathe : 1
life : 1
into : 1
characters. : 1
for : 1
example, : 1
friend : 1
henri : 1
latouche : 1
had : 1
a : 3
good : 1
knowledge : 1
hanging : 1
wallpaper. : 1
transferred : 1
this : 1
pension : 1
vauquer : 1
le : 1
père : 1
goriot, : 1
making : 1
wallpaper : 1
speak : 1
identities : 1
those : 1
living : 1
inside. : 1
 : 1
some : 1
critics : 1
consider : 1
writing : 1
exemplary : 1
naturalism—a : 1
more : 1
pessimistic : 1
analytical : 1
form : 1
realism, : 1
which : 1
seeks : 1
explain : 1
behavior : 1
as : 1
intrinsically : 1
linked : 1
with : 1
environment. : 1
french : 1
```

```
émile : 1
zola : 2
declared : 1
father : 1
naturalist : 2
novel.[96] : 1
indicated : 1
whilst : 1
romantics : 1
saw : 1
world : 1
colored : 1
lens, : 1
sees : 1
clear : 1
glass—precisely : 1
sort : 1
effect : 1
attempted : 1
achieve : 1
works. : 1
```

Optional Example: JSON and Dictionaries

Seeing all the keys and values

In the first example, we use `json.load()` to load a .json file. Compare the structure of the source file with the output. This file has a "root" (champagnes).

In [33]:
```python
import json

with open("orders.json") as f:
    temp = json.load(f)
    pairs = temp.items()

i = 0
for key, value in pairs:
    print(key, value)
```

```
champagnes [{'vintner': 'Dom Perignon', 'price': 115.67, 'vintage': ['1954',
'1979', '2000'], 'available': False, 'delivery': True, 'supplier': {'name': '
Lars Ingersol', 'phone': None, 'email': 'janedoe@email.com'}}, {'vintner': 'V
euve Cliquot', 'price': 56.0, 'brut': True, 'available': True, 'delivery': Tr
ue, 'supplier': {'name': 'vins Benoît', 'phone': '033-33-19-32', 'email': 'vi
ns@benoit.fr'}}]
```

In [34]:
```python
# Example as above but the source has -no- root

import json

with open("contacts.json") as f:
    temp = json.load(f)
    pairs = temp.items()

for key in pairs:
    print(key)
```

```
('Alice', {'phone': '2341', 'addr': '87 Eastlake Court'})
('Beth', {'phone': '9102', 'addr': '563 Hartford Drive'})
('Randy', {'phone': '4563', 'addr': '93 SW 43rd'})
```

> ## Json hard-coded string
>
> using `json.loads()` and not `load()` . Note the difference. Demos the idea of the `main` .

In [35]:
```python
import json

def main():
    # create a simple JSON array
    jsonString = '{"Shakespeare":"Hamlet","Kipling":"Kim","Ricoeur":"From tex

    # change the JSON string into a JSON object
    jsonObject = json.loads(jsonString)

    # print the keys and values
    for key in jsonObject:
        value = jsonObject[key]
        print("The key: {} and value: {}".format(key, value))

if __name__ == '__main__':
    main()
```

```
The key: Shakespeare and value: Hamlet
The key: Kipling and value: Kim
The key: Ricoeur and value: From text to action
```

Checking for a key in the data

When ingesting data, better to check what's there than to assume it is.

In [36]:
```
import json

# if the json is IN the script, use triple-quotes or single-quote
musicJson ="""{
   "id": 43,
   "composer": "Camille Saint-Saëns",
   "piece": "Symphony #3, with organ",
   "runtime": 601643,
   "email": "jhon@pynative.com"
}"""


musicians = json.loads(musicJson)
if "runtime" in musicians:
    print("Key exists: ", musicians["piece"], " runs ",musicians["runtime"],
else:
    print("No runtime data available.")
```

```
Key exists:  Symphony #3, with organ  runs  601643 seconds.
```

Using if statements ...

In [37]:
```
import json

myContacts = """
{
   "id": 1,
   "name": "明娃",
   "area": "zh",
   "email": "ming@wa.com"
}
"""

contact = json.loads(myContacts)
if not (contact.get("email") is None):
        print("Email for ", contact.get("name"), contact.get("email"))
else:
        print("No email value is listed.")
```

```
Email for  明娃 ming@wa.com
```

Nested key

Most of the data exported from SQL and the like include a root and nested keys. This example prints a nested key directly.

In [38]:
```
import json

studentJson = """{
   "class":{
      "student":{
         "name":"Amelia Magro",
         "grades":{
            "french":70,
            "mathematics":100
         }
      }
   }
}"""
sampleDict = json.loads(studentJson)
if 'grades' in sampleDict['class']['student']:
    print(sampleDict['class']['student']['name'], "grades are")
    print(sampleDict['class']['student']['grades'])
```

```
Amelia Magro grades are
{'french': 70, 'mathematics': 100}
```

Checking if the root is there ...

In [39]:
```
import json

studentJson = """{
   "class":{
      "student":{
         "name":"Amelia Magro",
         "grades":{
            "french":70,
            "mathematics":100
         }
      }
   }
}"""

print("Direct access to see if your grades key is here ... ")
sampleDict = json.loads(studentJson)
if 'class' in sampleDict:
    if 'student' in sampleDict['class']:
        if 'grades' in sampleDict['class']['student']:
            print(sampleDict['class']['student']['grades'])
        else:
            print("Sorry, that key isn't here.")
```

```
Direct access to see if your grades key is here ...
{'french': 70, 'mathematics': 100}
```

What if ... individual sets of data as dictionaries with a root?

JSON key contains a value in the form of an array or dictionary. In this case, if you need all values, we can iterate the nested JSON array. The exported data must have unique element names, here x1 and x2 (as opposed to x).

```
In [40]: import json

studentJson = """{
   "class":{
      "student1":{
         "name":"Amelia Magro",
         "grades":{
            "french": 70,
            "mathematics": 100
         },
         "subjects": ["Français 132", "Discrete Math", "Quantum Chemistry", "
      },
      "student2":{
         "name":"Noé Benoît",
         "grades":{
            "french": 100,
            "mathematics": 85
         },
         "subjects": ["Français 300", "Deutsche Literatur", "Soviet Era", "Py
      }
   }
}"""
sampleDict = json.loads(studentJson)
if 'class' in sampleDict:
    if 'student1' in sampleDict['class']:
        if 'subjects' in sampleDict['class']['student1']:
            print("We found your inquiry for", sampleDict['class']['student1'
            for sub in sampleDict['class']['student1']['subjects']:
                print(sub)
```

```
We found your inquiry for Amelia Magro
Français 132
Discrete Math
Quantum Chemistry
Fields
```

Nested sets

Most of the time data exported from SQL will use the name of the table as the "root" of the .json tree. It's not uncommon when converting from .xml to .json, too, to have the plural form of the a set of data (here "books") be the root for individual items ("book"). This can be a pain for us to process. This demo is a perhaps klunky workaround. Are there better python solutions?

In [41]:
```
import json

books = [
    {
        "book" : {
            "author": "Balzac",
            "type": "pdf",
            "title": "Les amis",
            "pub_year": 1888
        }
    },
    {
        "book" : {
            "author": "Voltaire",
            "type": "pdf",
            "title": "Pantagruel",
            "pub_year": 1725
        }
    },
    {
        "book" : {
            "author": "Ленин",
            "type": "trash",
            "title": "Что делать?",
            "pub_year": 1913
        }
    },
    {}  # an empty {}
]

print(books)
print("-"*50)


class DictQuery(dict):
    def get(self, path, default = None):
        keys = path.split("/")
        val = None

        for key in keys:
            if val:
                if isinstance(val, list):
                    val = [ v.get(key, default) if v else None for v in val]
                else:
                    val = val.get(key, default)
            else:
                val = dict.get(self, key, default)

            if not val:
                break;

        return val

for item in books:
    print(DictQuery(item).get("book/author"), DictQuery(item).get("book/title
```

```
[{'book': {'author': 'Balzac', 'type': 'pdf', 'title': 'Les amis', 'pub_year
': 1888}}, {'book': {'author': 'Voltaire', 'type': 'pdf', 'title': 'Pantagrue
l', 'pub_year': 1725}}, {'book': {'author': 'Ленин', 'type': 'trash', 'title
': 'Что делать?', 'pub_year': 1913}}, {}]
--------------------------------------------------
Balzac Les amis
Voltaire Pantagruel
Ленин Что делать?
```

Saving json

We can dump a dictionary into a .json file. But for humans, it's nice to make it pretty.

In [42]:
```python
# print to screen, pretty.  Change the indent to compare output.
import json

my_dictionary = { "book" : {
    "author": "Balzac",
    "type": "pdf",
    "title": "Les amis",
    "pub_year": 1888
    }
},
{
"book" : {
    "author": "Voltaire",
    "type": "pdf",
    "title": "Pantagruel",
    "pub_year": 1725
}
},
{
"book" : {
    "author": "Ленин",
    "type": "self-justified evil",
    "title": "Что делать?",
    "pub_year": 1913
    }
}

# TO SCREEN (STDOUT)
saving_myBooks = json.dumps(my_dictionary, indent = 4)
print(saving_myBooks)

# TO A FILE; PLEASE SAVE AS utf-8
# and ust in case for windows ...
with open("bookstoredump.json", "w", encoding="utf-8") as f:
    json.dump(my_dictionary, f, ensure_ascii = False, indent = 4)
```

```
[
    {
        "book": {
            "author": "Balzac",
            "type": "pdf",
            "title": "Les amis",
            "pub_year": 1888
        }
    }
```

]

This completes the notebook version for Week 3.

GB, subject to colleagues' edits.