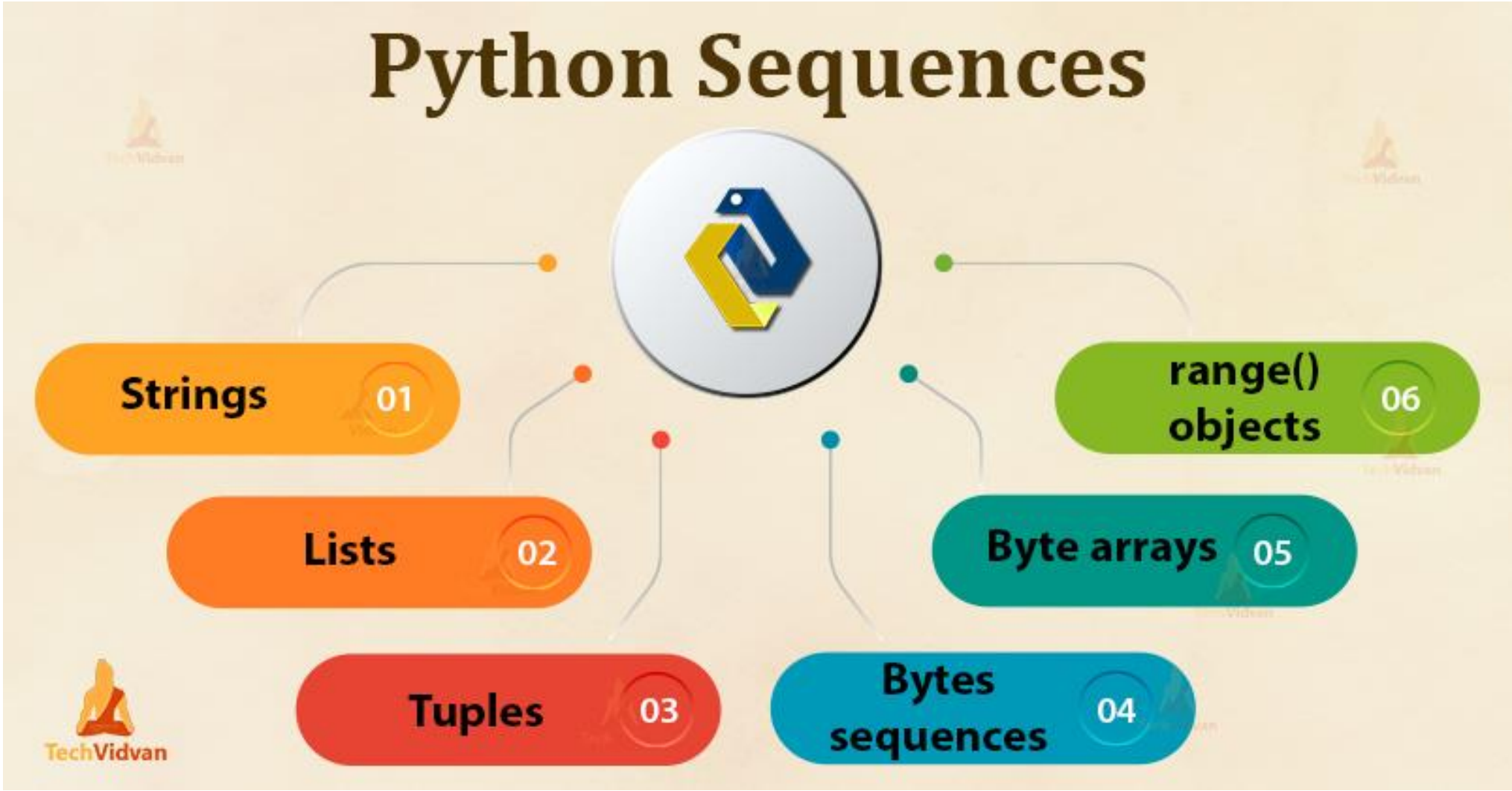
Python Intermediate Programming

Unit 0: Introduction

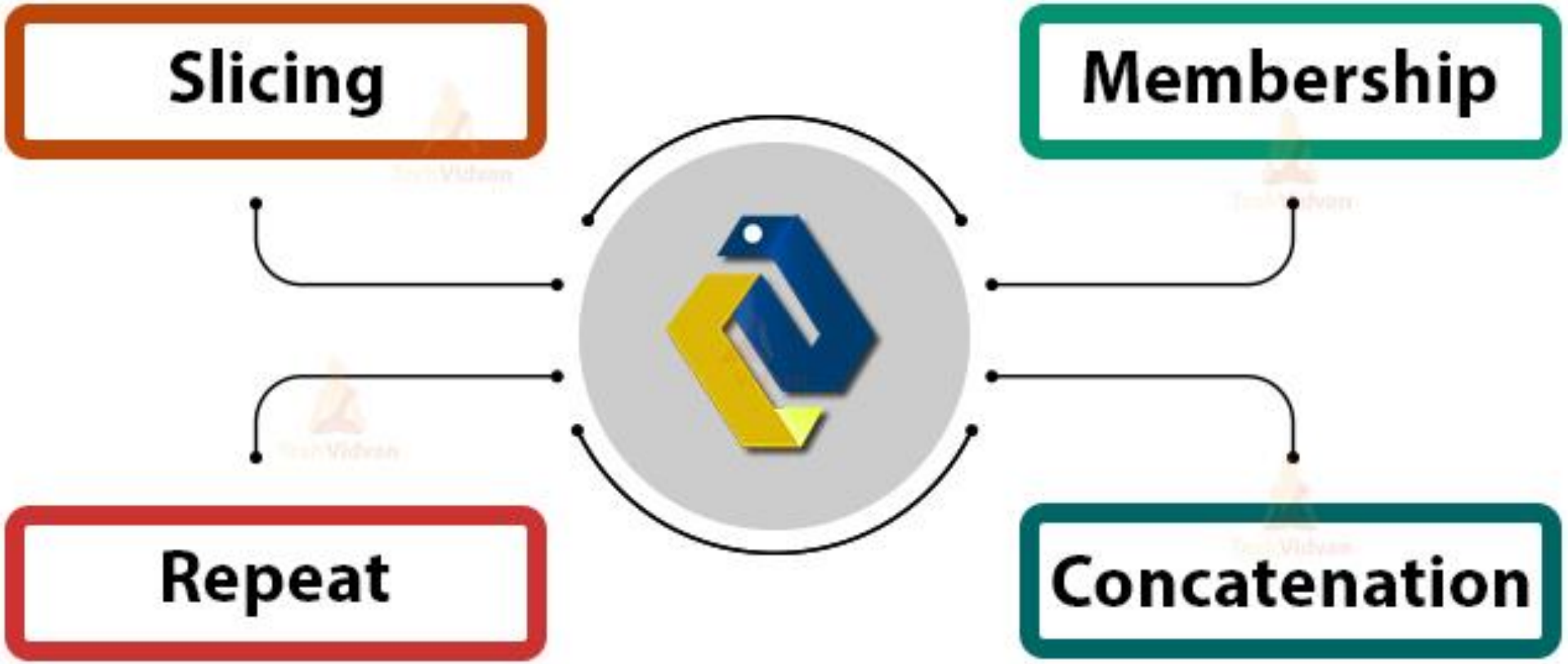
PYTHON REVIEW B: DATA SEQUENCES AND ORDERING

DR. ERIC CHOU

IEEE SENIOR MEMBER


Python Sequences
Strings 01
Lists 02
Tuples 03
Bytes sequences 04
Byte arrays 05
range() objects 06
TechVidvan

Operations on Python Sequences



Iterable

LECTURE 1

Iterable

- When we specify that an argument of a function is iterable, it might be one of the standard data structures in Python: str, list, tuple, set, dict. All these data structures are iterable: we can iterate over the values they contain by using a simple for loop.
- But we will learn other Python features (classes and generators) that are also iterable. The difference is, that for standard Python data structures we can call the len function on them and index them [...]; but for general iterable arguments we CANNOT call `len` nor `index` them: we can only iterate over their value with a for loop: getting the first value, the second value, etc. (although later in the quarter we will learn how to call iter and next explicitly on iterables; for loops implicitly call these two functions on iterables).

Iterable

- Also, we can always use a comprehension (discussed later in this lecture) to transform any iterable into a list of its values, doing so takes up extra time and space, and should be avoided unless necessary. But learning how to convert an iterable into a list is instructive.

sort (a list method)/ sorted (a function)

LECTURE 2

sort / sorted

- First, we will examine the following similar definitions of sort/sorted. Then we will learn the differences between these similar and related features below.
  1) sort is a method defined on arguments that are LIST objects; it returns None but MUTATES its list argument to be in some specified order: e.g., alist.sort() or alist.sort(reverse=True)
  2) sorted is a function defined on arguments that are any ITERABLE object; it returns a LIST of its argument's values, in some specified order. The argument itself is NOT MUTATED: e.g., sorted(adict) or sorted(adict,reverse=True). Neither changes adict to be sorted, because dictionaries are never sorted; both produce a LIST of sorted keys in adict.
- So, the sort method can be applied only to lists, and the sorted function can be applied to any iterable (str, lists, tuples, sets, dict, etc.).

sort / sorted

- For example, if votes is the list of 2-tuples (candidates, votes) below, we can execute the following code.

```
votes=[('Charlie',20), ('Able',10), ('Baker',20), ('Dog',15)]
votes.sort()
for c,v in votes:      # note parallel/unpacking assignment
    print('Candidate',c,'received',v,'votes')
print(votes)
```

- The call votes.sort() uses the sort METHOD to sort the LIST (MUTATE it); then the for loop iterates over this newly-ordered list and prints the information in the list in the order it now appears. When the entire votes list is printed after the loop, we see the list has been MUTATED and is now sorted.

sort / sorted

- Contrast this with the following code.

```
votes=[('Charlie',20),('Able',10),('Baker',20),('Dog',15)]
for c,v in sorted(votes):# parallel/unpacking
                         # assignment for key:value
    print('Candidate',c,'received',v,'votes')
print(votes)
```

- Here we never sort/mutate the votes list. Instead we use the sorted FUNCTION, which takes an ITERABLE (lists are iterable) as an argument and returns a NEW LIST that is sorted. Then we iterate over that returned list to print its information (which is a sorted version of votes). But here, when we print the votes list at the end, we see that the list remained unchanged.

sort / sorted

- The sorted function can be thought of as creating a list by iterating over the parameter, sorting that list, and then returning the sorted list: sorted mutates the list it creates internally, not the argument matching iterable.
- Think of the sorted function as defined by

```
def sorted (iterable):
  alist=list(iterable) # create a list with all the iterable's values
  alist.sort()         # sort that list
  return alist         # return the sorted list
```

Question

- What would the following code do? If you understand the definitions above, you won't be fooled and the answer won't suprise you.

Hint: what does votes.sort() return?

```
votes=[('Charlie',20),('Able',10),('Baker',20),('Dog',15)]
for c,v in votes.sort():
    print('Candidate',c,'received',v,'votes')
print(votes)
```

sort / sorted

- If we were going to print some list in a sorted form many times, it would be more efficient to sort/mutate it once, and then use a regular for loop on that sorted list (either mutate the original or create a new, sorted list). But if we needed to keep the list in a certain order and care about space efficiency and/or don't care as much about time efficienty, we would not sort/mutate the list and instead call the sorted function whenever we need to process the list in a sorted order.
- Note that if we store votes_dict as a dict data structure (a dictionary associating candidates with the number of votes they received), and then tried to call the sort method on it, Python would raise an exception

```
votes_dict={'Charlie': 20, 'Able': 10, 'Baker': 20, 'Dog': 15}

votes_dict.sort()
```

sort / sorted

- It would show as

```
AttributeError: 'dict' object has no attribute 'sort'
```

- The problem is: the sort method is defined only on list class objects, not dict class objects.
- So, Python cannot execute votes_dict.sort()! It makes no sense to sort a dictionary. In fact, we cannot sort strings (they are immutable); we cannot sort tuples (their order is immutable); we cannot sort sets (they have no order, which actually allows them to operate more efficiently; we'll learn why later in the quarter); we cannot sort dicts (like sets, they have no order, which allows them to operate more efficiently; ditto).

sort / sorted

- BUT, we can call the sorted function on all four of these data structures, and generally on anything that is iterable: Python executes sorted by creating a temporary list from all the values produced by the iterable, then sorts that list, and then returns the sorted list. Here is one example of how the sorted function processes votes_dict. Note that executing sorted(votes_dict) is the same as executing sorted(votes_dict.keys()) which produces and iterates over a sorted list of the dict's keys.

```
votes_dict={'Charlie':20, 'Able': 10, 'Baker':20, 'Dog':15}
for c in sorted(votes_dict):
    #same as: for c in sorted(votes_dict.keys())
    print('Candidate',c,'received',votes_dict[c],'votes')
print(votes_dict)
```

Alternative Example

- Note: if we wrote

```
votes_dict={'Charlie':20, 'Able':10, 'Baker':20, 'Dog':15}
print(sorted(votes_dict))
```

- Python prints a list, returned by sorted, of the dictionary's keys in sorted order:

```
['Able', 'Baker', 'Charlie', 'Dog']
```

sort / sorted

- Well, this is one "normal" way to iterate over a sorted list built from a dictionary. We can also iterate over sorted "items" in a dictionary as follows (the difference is in the for loop and the print). We will examine more about dicts and the different ways to iterate over them later in this lecture. Recall that each item in a dictionary is a 2-tuple consisting of one key and its associated value.

```
votes_dict={'Charlie':20, 'Able': 10, 'Baker':20, 'Dog':15}
for c,v in sorted(votes_dict.items()):
    # note parallel/unpacking assignment
    print('Candidate',c,'received',v,'votes')
print(votes_dict)
```

Alternative Example

Note: if we wrote

```
votes_dict={'Charlie':20, 'Able':10, 'Baker':20, 'Dog':15}
print(list(votes_dict.items()))
print(sorted(votes_dict.items()))
```

- Python first prints a list of the dictionary's items in an unspecified order, then it prints a list of the smae dictionary's items, in sorted order:

```
[('Able', 10), ('Dog', 15), ('Charlie', 20), ('Baker', 20)]
[('Able', 10), ('Baker', 20), ('Charlie', 20), ('Dog', 15)]
```

both list(...) and sorted(...) produce a list of the items in the dictionary, with only sorted(....) producing a sorted list.

sort / sorted

- Notice that this print doesn't access votes[c] to get the votes: that is the second item in each 2-tuple being iterated over using .items(). This is because iterating over votes_dict.items() produces a sequence of 2-tuples, each containing one key and its associated value. The order that these 2-tuples appear in the list is unspecified, but using the sorted function ensures that the keys are examined in order.

sort / sorted

- How does sort/sorted work? How do they know how to compare the values they are sorting? There is a standard way to compare any data structures, but we can also use the "key" and "reverse" parameters (which must be used with their names, not positionally) to tell Python how to do the sorting. The reverse parameter is simpler, so let's look at it first; writing sorted(votes,reverse=True) sorts, but in the reverse order (writing reverse=False is like not specifying reverse at all). So, returning to votes as a list,

```
votes=[('Charlie',20), ('Able',10), ('Baker',20), ('Dog',15)]
print(sorted(votes,reverse=True))
```

prints

```
[('Dog', 15), ('Charlie', 20), ('Baker', 20), ('Able', 10)]
```

sort / sorted

- What sort is doing is comparing each value in the list to the others using the standard meaning of <. You should know how Python compares str values, but how does Python compare whether one tuple is less than another? Or whether one list is less than another? The algorithm in analagous to strings, so let's first re-examine comparing strings. The ordering, by the way, is called "lexicographic ordering", and also dictionary ordering (related to the order words appear in dictionaries that are books, not Python dicts/dictionaries).

Comparing Strings

LECTURE 3

Comparing Strings in Python:

String comparison: x to y (high level):

- Find the first/minimum index i such that the ith character in x and y are different (e.g., x[i] != y[i]). If that character in x is less than that character in y (by the standard ASCII table) then x is less than y; if that character in x is greater than that character in y then x is greater than y;
- if there is no such different character, then x compares to y as x's length compares to y's (either less than, equal or greater than).

Comparing Strings in Python:

Here are three examples:

1) How do we compare x = 'aunt' and y = 'anteater'? The first/minimum i such that the characters are different is 1: x[1] is 'u' and y[1] is 'n'; 'u' is greater than 'n' so x > y.
2) How do we compare x = 'ant' and y = 'anteater'? There is no first/minimum i such that the characters are different; len(x) < len(y) so x < y. The word 'ant' appears in an English dictionary before the word 'anteater' ('ant' is a prefix of 'anteater').
3) How do we compare x = 'ant' and y = 'ant'? There is no first/minimum i such that the characters are different; len(x) == len(y) so x == y. I show this example, which is trivial, just to be complete.

Comparing Strings in Python:

- See the Handouts(General) webpage showing the ASCII character set, because there are some surprises. You should memorize that the digits and lower/upper case letters compare in order, and all digits < all upper-case letters < all lower-case letters. So 'TINY' < 'huge' is True because 'T' is < 'h' (all upper-case letters have smaller ASCII values than any lower-case letters).
- Likewise, 'Aunt' < 'aunt' because 'A' < 'a'. Note that we can always use the upper() method (as in x.upper() < y.upper()) or lower() method to perform a comparison that ignores the case of the letters in a string.
- Use these rules to determine whether '5' < '15'.

Comparing Strings in Python:

- Back to comparing tuples (or lists). We basically do the same thing. We look at what is in index 0 in the tuples; if different, then the tuples compare in the same way as the values in index 0 compare; if the values at this index are the same, we keep going until we find a difference, and compare the tuples the same way that the differences compare; if there are no differences, the tuples compare the same way their lengths compare.
- So ('UCI', 100) < ('UCSD', 50) is True because at index 0, the string values are different: 'UCI' < 'UCSD' (because index 2 in these strings are different, and 'I' < 'S'). Whereas ('UCI', 100) < ('UCI', 200) is True because the values at index 0 are the same ('UCI' == 'UCI'), so we go to index 1, where we find different values, and 100 < 200. Finally, ('UCI', 100) < ('UCI', 100, 'extra') is True because the values at index 0 are the same ('UCI' == 'UCI') and the values at index 1 are the same (100 == 100), and the last tuple has a larger length.

Comparing Strings in Python:

- Recall the sorting code from above.

```
votes=[('Charlie',20), ('Able',10), ('Baker',20), ('Dog',15)]
for c,v in sorted(votes):
    print('Candidate',c,'received',v,'votes')
print(votes)
```

- The reason that the values come out in the order they do in the code above is because the names that are in the first index in each 2-tuple are different and ensure the tuples are sorted alphabetically. Python never gets to looking at the second value in each tuple, because the first values (the candidate names) are always different.

Comparing Strings in Python:

- Now, what if we don't want to sort by the natural tuple ordering. We can specify a "key" function that computes a key value for every value in what is being sorted, and the computed key values are used for comparison, not the original values themselves. These are the "keys" for comparison.
- See the by_vote_count function below; it takes a 2-tuple argument and returns only the second value in the tuple (recall indexes start at 0) for the key on which Python will compare. For the argument tuple ('Baker' ,20) by_vote_count returns 20.

```
def by_vote_count(t):
    return t[1] # remember t[0] is the first index, t[1] the second
```

Comparing Strings in Python:

- So, when we sort with key=by_vote_count, we are telling the sorted function to determine the order of values by calling the by_vote_count function on each value: so in this call of sorted, we are comparing tuples based solely on their vote part. So writing

```
votes=[('Charlie', 20), ('Able', 10), ('Baker' ,20), ('Dog', 15)]
for c,v in sorted(votes, key=by_vote_count):
    print('Candidate',c,'received',v,'votes')
```

produces

```
Candidate Able received 10 votes
Candidate Dog received 15 votes
Candidate Charlie received 20 votes
Candidate Baker received 20 votes
```

Comparing Strings in Python:

- First, notice that by writing "key=by_vote_count" Python didn't CALL the by_vote_count function (there are no parentheses) it just passed its associated function object to the key parameter in sorted. Inside the sorted function, by_vote_count's function object automatically is called where needed, to compare two 2-tuples, to determine in which order these values will appear in the returned list.
- Also, because Charlie and Baker both received the same number of votes, they both appear at the bottom, in an unspecified order (equal values can appear in any order).

Comparing Strings in Python:

Also notice that the tuples are printed in ascending votes; generally, for elections we want the votes to be descending, so we can combine using key and reverse (with reverse=True) in the call to the sorted function.

```
votes=[('Charlie',20), ('Able',10), ('Baker',20), ('Dog',15)]
for c,v in sorted(votes, key=by_vote_count, reverse=True):
    print('Candidate',c,'received',v,'votes')
```

which produces

```
Candidate Charlie received 20 votes
Candidate Baker received 20 votes
Candidate Dog received 15 votes
Candidate Able received 10 votes
```

Comparing Strings in Python:

Again, since the top two candidates both received the same number of votes, they can appear in any order: the order is unspecified.

Now, rather than define this simple `by_vote_count` function, we can use a lambda instead, and write the following.

…

```
for c,v in sorted(votes, key=(lambda t : t[1]), reverse=True):
```

…

So, now we don't have to write/name that extra by_vote_count function. Of course, if we did write it, we could reuse it wherever we wanted, instead of rewriting the lambda (but the lambdas are pretty small). By writing the lambda, the code is inside the call to sorted: we don't have to go look for some function defintion. So, we now know how to use functions and lambdas in sorted.

Comparing Strings in Python:

- They are used the same way in sort. If votes is the list shown above, we can call votes.sort(key=(lambda t : t[1]), reverse=True) to sort this list, mutating it.
- Another way to sort in reverse order (for integers) is to use the "negate" trick illustrated below.

```
...
for c,v in sorted(votes, key=(lambda t : -t[1]) ):
...
```

- Here, we have negated the vote count part of the tuple, and removed reverse=True. So it is sorting from smallest to largest, but by the negative of the vote values (because that is what key says to do). It is using -20, -10, -20, and -15 to sort. So here the biggest vote count corresponds to the smallest negative number (so that tuple will appear first). The tuples appear in the order specified by the key unctions: -20, -20, -15, -10 (smallest to largest). These negative values returned by the key function are used only to determine the order, they do not appear in the results.

Comparing Strings in Python:

- Finally, typically when multiple candidates are tied for votes, we print their names close together (because they all have the same number of votes) but also in alphabetical order. We do this in Python by specifying a key function that returns a tuple in which the vote count is checked first and sorted in descending order; and only if they are tied will the names be checked and sorted in ascending order. Because we want the tuples in decreasing vote counts but increasing names, we cannot use reverse=True: it would reverse both the vote AND name comparisons; we need to resort to the "negation trick" above and write

```
votes=[('Charlie',20), ('Able',10), ('Baker',20), ('Dog',15)]
for c,v in sorted(votes, key=(lambda t : (-t[1],t[0])) ):
    print('Candidate',c,'received',v,'votes')
```

Comparing Strings in Python:

- So, it compares the 2-tuples (-20, 'Charlie'), (-10, 'Able'), (-20, 'Baker'), and (-15, 'Dog') when ordering the actual 2-tuples, and by what we have learned

```
(-20,'Baker') < (-20,'Charlie') < (-15,'Dog') < (-10,'Able')
```

which produces

```
Candidate Baker received 20 votes
Candidate Charlie received 20 votes
Candidate Dog received 15 votes
Candidate Able received 10 votes
```

So with this key function, the previous order is guaranteed, even though Baker and Charlie both received the same number (20) of votes.

Comparing Strings in Python:

- Finally, note that the lambda in the key ensures Python compares ('Charlie', 20) and ('Dog', 15) as if they were (-20, 'Charlie') and (-15, 'Dog'), so the first tuple will be less and appear earlier in the sorted list (the one with the highest votes has the lowest negative votes).
- And when Python compares ('Charlie', 20) and ('Baker' ,20) as if they were (-20, 'Charlie') and (-20, 'Baker') so the second tuple will be less and the tuple it was produced from will appear earlier (equal votes and then 'Baker' < 'Charlie').

Comparing Strings in Python:

So, think of sorting

```
('Charlie', 20)    ('Able', 10)    ('Baker' ,20)    ('Dog', 15)
        |                |               |                |
        |                |               |                |  using the key function
        V                V               V                V
(-20, 'Charlie')   (-10, 'Able')   (-20, 'Baker')   (-15, 'Dog')
```

which sorts to

```
(-20,'Baker')      (-20,'Charlie')    (-15,'Dog')     (-10,'Able')
        |                |               |                |
        |                |               |                | undo the key function
        V                V               V                V
('Baker',20)       ('Charlie', 20)    ('Dog',15)      ('Able', 10)
```

So, the order of the actual list returned by sorted is

```
[('Baker', 20), ('Charlie', 20), ('Dog', 15),  ('Able', 10)]
```

which is sorted by decreasing vote, with equal votes sorted alphabetically increasing by name.

Comparing Strings in Python:

- Our ability to use this "negation trick" works in SOME cases, but unfortunately NOT IN ALL cases: we can sort arbitrarily exactly when "all non-numerical data is sorted in the same way" (all increasing or all decreasing). In such cases, we can negate any numerical data, if we need to sort it decreasing. So, we can sort the above example because we are sorting only one non-numerical datum. By the requirement, all non-numerical data (there is only one) is sorted the same way (increasing).
- Using the "negation trick" is therefor not generalizable to all sorting tasks: for example, if we had a list of 2-tuples containing strings of candidates and strings of the states they are running in

```
[('Charlie','CA'),('Able','NY'),('Baker','CA'),('Dog','IL')]
```

Comparing Strings in Python:

- there is no way to use the "negation trick" to sort them primarily bydecreasing state, and secondarily by increasing name (when two candidates come from the same state). Here it is not the case that "all non-numerical data is sorted the same way": each string data is sorted differently. In this case the sorted list would be

```
[('Able','NY'),('Dog','IL'),('Charlie','CA'),('Baker','CA')]
```

- We cannot apply the "negation trick" to either of the strings: we cannot negate strings at all. But we can produce a list in this order calling sorted multiple times, as is illustrated below.

Arbitrary Sorting

LECTURE 4

Arbitrary Sorting

multiple calls to sorted with "stable" sorting

Python's sorted function (and sort method) are "stable". This property means that "equal" values (decided naturally, or with the key function supplied) keep their same "relative" order (left-to-right positions) in the data being sorted.

For example, assume that db is a list of 2-tuples, each specifying a student: index 0 is a student's name; index 1 is that student's grade.

```
db = [('Bob','A'),    ('Mary','C'), ('Pat','B'), ('Fred','B'), ('Gail','A'),

      ('Irving','C'), ('Betty','B'), ('Rich','F')]
```

If we call

```
sorted(db, key = lambda x : x[1])   # sorted by index #1 only: their grade
```

Arbitrary Sorting

multiple calls to sorted with "stable" sorting

- Python returns the following list, whose 2-tuples are sorted by grade. Because sorting is "stable", all 2-tuples with the same grade (equal values by the key function) keep their same relative order (left-to-right positions) in the list.

```
[('Bob','A'),  ('Gail','A'), ('Pat','B'),
 ('Fred','B'), ('Betty','B'),('Mary','C'),
 ('Irving','C'), ('Rich','F')]
```

Notice that...

1. in the original list, the students with 'A' grades are 'Bob' and 'Gail', with 'Bob' to the left of 'Gail'. In the returned list, 'Bob' is also to the left of 'Gail'.
2. in the original list, the students with 'B' grades are 'Pat', 'Fred', and 'Betty', with 'Pat' to the left of 'Fred' and 'Fred' to the left of 'Betty'. In the returned list, 'Pat' is to the left of 'Fred' and 'Fred' is to the left of 'Betty'.
3. in the original list, the students with 'C' grades are 'Mary' and 'Irving', with 'Mary' to the left of 'Irving'. In the returned list, 'Mary' is also to the left of 'Irving'.
4. there is only one student with an 'F' grade.

Sorting with Lambda Comparators

- Now, suppose that we wanted to sort this list so that primarily the 2-tuples are sorted DECREASING by grade ('F's, then 'D's, then 'C's, then 'B's, then 'A's); and for students who have equal grades, the 2-tuples are sorted INCREASING alphabetically by student name. We can accomplish this task by writing

```
sorted( sorted(db, key=lambda x : x[0]),
        key=lambda x : x[1], reverse = True )
```

- The inner call to sorted produces the list

```
[('Betty','B'), ('Bob','A'), ('Fred','B'),
 ('Gail','A'), ('Irving','C'), ('Mary','C'),
 ('Pat','B'), ('Rich','F')]
```

which is sorted INCREASING by the student's name: all names are distinct (different), so stability is irrelevant here.

Sorting with Lambda Comparators

- The outer call to sorted, using the original db list as an argument, produces the list

```
[('Rich','F'), ('Irving','C'), ('Mary','C'),
 ('Betty','B'), ('Fred','B'), ('Pat','B'),
 ('Bob','A'),    ('Gail','A')]
```

which is sorted DECREASING by the student's grade: for equal grades, the stability property ensures that all names are still sorted INCREASING by the student's name (because the argument is sorted that way). Here, the 'B' students are in the alphabetical order 'Betty', 'fred', and 'Pat'. Note that we could simplify this code to just

```
sorted( sorted(db), key=lambda x : x[1], reverse = True )
```

Sorting with Lambda Comparators

- because for the inner call to sorted, no key function is the same as the key function that specifies sorting the 2-tuples in the natural way. We could use a temporary variable and write this code as

```
temp = sorted(db)
sorted( temp, key = lambda x : x[1], reverse = True )
```

- Thus, we can sort complex structure in any way (some data increasing, some data decreasing) by calling sorted multiple times on it. The LAST call will dictate the primary order in which it is sorted; each preceding calls dictates how the data is sorted if values specified by the earlier orders are equal.
- Expeiment.

Sorting with Lambda Comparators

- Finally, to solve the original problem above (with candidates and their states: also involving two strings sorted primarily DECREASING on state and INCREASING on name)

```
db=[('Charlie','CA'),('Able','NY'),('Baker','CA'),('Dog','IL')]
```

we would call

```
sorted( sorted(db), key=lambda x : x[1], reverse = True )
```

- Bottom line on sorting:
- To achieve our sorting goal, if we can call sort once, using a complicated key (possibly involving the "negation trick") and reverse, that is the preferred approach: it is shortest/clearest in code and fastest in computer time. But if we are unable to specify a single key function and reverse that do the job, we can always call sort multiple times, using multiple key functions and multiple reverses, sometimes relying on the "stability" property to achieve our sorting goal.
- We now have a general tool bag for sorting all types of information.

The print function

LECTURE 5

The print function

- Notice how the **sep** and **end** parameters in print help control how the printed values are separated and what is printed after the last one. Recall that print can have any number of arguments (we will see how this is done in Python soon), and it prints the str(...) of each parameter. By default, **sep=' ' (space)** and **end='\n' (newline)**. So, the following

```
print(1,2,3,4,sep='--',end='/')
print(5,6,7,8,sep='x',end='**')
```

prints

```
1--2--3--4/5x6x7x8**
```

The print function

- Also recall that all functions must return a value. The print function returns the value None: this function serves as a statement: it has an "effect" (of displaying information in the console window; we might say changing the state of the console) but returns no useful "value"; but all functions must return a value, so this one returns None.
- Sometimes we use **`sep=''`** to control spaces more finely. In this case we must put in all the spaces ourselves. If we want to separate only 2 and 3 by a space, we write

```
print(1,2,' ',3,sep='')
```

which prints

```
12 3
```

The print function

- Still, other times we can concatenate values together into one string (which requires us to explicitly use the str function on the things we want to print).

```
x = 10
print('Your answer of '+str(x)+' is too high.'+
      '\nThis is on the next line')
```

- Note the use of the "escape" sequence \n to generate a new line.

The print function

- Finally, we can also use the very general-purpose .format function. This function is illustrated in Section 6.1.3 in the documentation of The Python Standard Library. The following two statements print equivalently: for a string with many format substitutions, I prefer the form with a name (here x) in the braces of the string and as a named parameter in the arguments to format.

```
print('Your answer of {} is too high\nThis is on the next line'.format(10))
```

```
print('Your answer of {x} is too high\nThis is on the next line'.format(x=10))
```

String/List/Tuple (SLT) slicing

LECTURE 6

String/List/Tuple (SLT) slicing

- SLTs represent sequences of indexable values, whose indexes start at index 0, and go up to -but do not include- the length of the SLT (which we can compute using the len function).

String/List/Tuple (SLT) slicing

1) **Indexing:** We can index an SLT by writing SLT[i], where i is in the range 0 to len(SLT)-1 inclusive, or i is negative and i is in the range -len(SLT) to -1 inclusive: if an index is not in these ranges, Python raises an exception. Note SLT[0] is the value store in the first index, SLT[-1] is the value stored in the last index, SLT[-2] is the value stored in the 2nd to last index.

2) **Slicing:** We can specify a slice by SLT[i:j] which includes SLT[i] followed by SLT[i+1], ... SLT[j-1]. Slicing a string produces another string, slicing a list produces another list, and slicing a tuple produces another tuple. The resulting structures has j-i elements if indexes i through j are in the SLT (or 0 if j-i is <= 0); for this formula, both indexes must be non-negative or both positive; if they are different, convert the negative to it non-negative (or the non-negative to its negative) equivalent.

- If the first index come before index 0, then index 0 is used; if the second index comes after the biggest index, then index len(SLT) is used. Finally, if the first index is omitted it is 0; if the second index is omitted it is len(SLT).

String/List/Tuple (SLT) slicing

- Here are some examples

```
s = 'abcde'
x = s[1:3]
print (x)      # prints 'bc' which is 3-1 = 2 values
x = s[-4:-1]
print (x)      # prints 'bcd' same as s[1:4] which is 4-1 = 3 values
x = s[1:-2]
print (x)      # prints 'bc' same as s[1:3] which is 3-1 = 2
x = s[-4:-2]
print (x)      # prints 'bc' same as s[1:3] which is 3-1 = 2 values
x = s[1:10]
print (x)      # prints 'bcde' which is len(x)-1 = 4 values
```

String/List/Tuple (SLT) slicing

likewise

```
s = ('a','b','c','d','e')
x = s[1:3]
print (x)      # prints ('b','c') which is 3-1 = 2 values
x = s[-4:-1]
print (x)      # prints ('b','c','d') which is -1-(-4) = 3 values
         # s[:i] is index 0 up to but not including i
         # (can be positive or negative)
         # s[i:] is index i up to and including the last index;
         # so, s[-2:] is the last two values.
```

String/List/Tuple (SLT) slicing

3) Slicing with a stride: We can specify a slice by SLT[i:j:k] which includes SLT[i] followed by SLT[i+k], SLT[i+2k] ... SLT[j-1]. This follows the rules for slicing too, and allows negative numbers for indexing.

```
s = ('a','b','c','d','e')
x = s[::2]
print (x)      # prints ('a','c','e')
x = s[1::2]
print (x)      # prints ('b','d')

x = s[3:1:-1]
print (x)      # prints ('d','c')
x = s[-1:1:-1]
print (x)      # prints ('e','d','c')
```

String/List/Tuple (SLT) slicing

- When the stride is omitted, it is +1. If the stride is negative, if the first index is omitted it is `len(SLT)`; if the last index is omitted it is -1.
- Compare these to the values if omitted above in part 2, which assumed a positive stride.
- Experiment with various slices of strings (the easiest to write) to verify you understand what results they produce.

Conditional statement vs. Conditional expression

LECTURE 7

Conditional statement vs. Conditional expression

- Python has an if/else STATEMENT, which is a conditional statement. It also has a conditional EXPRESSION that uses the same keywords (if and else), which while not as generally useful, sometimes is exactly the right tool to simplify a programming task.
- A conditional statement uses a boolean expression to decide which indented block of statements to execute; a conditional expression uses a boolean expression to decide which of two other expressions to evaluate: the value of the evaluated expression is the value of the conditional expression.

Conditional Expression

- The form of a conditional expression is

```
resultT if test else resultF
```

- This says, the expression evaluates to the value of **`resultT`** if test is True and the value of **`resultF`** if test is False; first it evaluates test, and then evaluates either **`resultT`** or **`resultF`** (but only one, not the other) as necessary.
- Like other short-circuit operators in Python (do you know which?) it evaluates only the subexpressions it needs to determine the result.
- I often put conditional expressions inside parentheses for clarity (as I did for lambdas). See the examples below.

Conditional Expression

- Here is a simple example. We start with a conditional statement, which always stores a value into min: either x or y depending on whether x <= y. Note that regardless of the test, min is bound to some value.

```
if x <= y:
    min = x
else:
    min = y
```

- We can write this using a simpler conditional expression, capturing the fact that we are always storing into min, and just deciding which value to store.

```
min = (x if x <= y else y)
```

Conditional Expression

- Not all conditional statements can be converted into conditional expressions; typically, only simple ones can: ones with a single statement in their indented blocks. But using conditional expressions in these cases simplifies the code even more. So, attempt to use conditional expression, but use good judgement after you see what the code looks like.
- Here is another example; it always prints x followed by some message

```
if x % 2 == 0:
    print(x,'is even')
else:
    print(x,'is odd')
```

Conditional Expression

- We can rewrite it as calling print with a conditional expression inside deciding what string to print at the end.

```
print(x, ('is even' if x%2 == 0 else 'is odd'))
```

- We can also write it as a one argument print using concatenation:

```
print(str(x)+' is '+('even' if x%2 == 0 else 'odd'))
```

- Note that conditional expressions REQUIRE using BOTH THE if AND else KEYWORDS, along with the test boolean expression and the two expressions needed in the cases where the test is True or the test is False. Some conditional statements use just if (and no else)

The else: block-else option in for/while loops

LECTURE 8

The else: block-else option in for/while loops

- For and while looping statements are described as follows. The else: block-else is optional, and not often used. But we will explore its meaning here.

```
for_statement   <= for index(es) in iterable:
                       block-body
                   [else:
                       block-else]

while_statement <= while <bool-expression>:
                       block-body
                   [else:
                       block-else]
```

The else: block-else option in for/while loops

- Here are the semantics of else: block-else.

```
If the else: block-else option appears, and the loop
terminated normally, (not with a break statement)
then execute block-else.
```

- Here is an example that makes good use of the else: block-else option. This code prints the first/lowest value (looking at the values 0 to 100 inclusive) for which the function **`special_property`** returns True (and then breaks out of the loop); otherwise, it prints that no value in this range had this property: so, it prints exactly one of these two messages. Note you cannot run this code, because there is no **`special_property`** function: I'm using it for illustration only.

The else: block-else option in for/while loops

```
for i in irange(100):
    if special_property(i):
        print(i,'is the first value with the special property')
        break
else:
    print('No value in the range had the special property')
```

- Without the else: block-else option, the simplest code that I can write that has equivalent meaning is as follows.

The else: block-else option in for/while loops

```
found_one = False
for i in irange(100):
    if special_property(i):
        print(i,'is the first with the special property')
        found_one = True
        break
if not found_one:
    print('No value in the range had the special property')
```

- This solution requires an extra name (found_one), an assignment to set and reset the name, and an if statement. Although I came up with the example above, I have not used the else: block-else option much in Python. Most programming languages that I have used don't have this special feature, so I'm still exploring its usefulness. Every so often I have found an elegant use of this construct.

The else: block-else option in for/while loops

- Can you predict what would happen if I removed the break statement in the bigger code above? Run the code to check your answer.

Argument/Parameter Matching

LECTURE 9

Argument/Parameter Matching

- Let‘s explore the argument/parameter matching rules. First we classify arguments and parameters, according the options they include. Remember that arguments appear in function CALLS and parameters appear in function HEADERS (the first line in a function defintion).

Arguments

positional argument: an argument NOT preceded by the name= option

named argument: an argument preceded by the name= option

Parameters

name-only parameter: a parameter not followed by =default argument value

default-argument parameter: a parameter followed by =default argument value

Argument/Parameter Matching

- When Python calls a function, it must define every parameter name in the function‘s header, and bind to it the argument value object matching that parameter’s name (just like an assignment statement). In the rules below, we will learn exactly how Python matches arguments to parameters according to three criteria: positions, parameter names, and default arguments for parameter names. We will also learn how to write functions that can receive an arbitrary number of arguments.
- Here is a concise statement of Python‘s rules for matching arguments to parameters. The rules are applied in this order (e.g., once you reach M3 we cannot go back to M1).

Argument/Parameter Matching

M1. Match positional argument values in the call sequentially to the parameters named in the header's corresponding positions (both name-only and default-argument parameters are OK to match). Stop when reaching any named argument in the call, or the * parameter (if any) in the header.

M2. If matching a * parameter in the header, match all remaining positional argument values to it. Python creates a tuple that stores all these arguments. The parameter name (typically args) is bound to this tuple.

M3. Match named-argument values in the call to their like-named parameters in the header (both name-only and default-argument parameters are OK).

Argument/Parameter Matching

M4. Match any remaining default-argument parameters in the header, un-matched by rules M1 and M3, with their specified default argument values.

M5. Exceptions: If at any time (a) an argument cannot match a parameter(e.g., a positional-argument follows a named-argument) or (b) a parameter is matched multiple times by arguments; or if at the end of the process (c) any parameter has not been matched or (d) if a named-argument does not match the name of a parameter, raise an exception: SyntaxError for (a) and TypeError for (b), (c), and (d). These exceptions report that the function call does not correctly match its header by these rules.

Argument/Parameter Matching

[When we examine a **`**kargs`** as a parameter, we will learn what Python does when there are extra named arguments in a function call: names besides those of parameters: preview: it puts all remaing named arguments in a dictionary, with their name as the key and their value associated with that key). The parameter name (typically kargs or kwargs) is bound to this dictionary.]

- When this argument-parameter matching process if finished, Python defines, (in the function‘s namespace), a name for every parameter and binds each to the unique argument it matches using the above rules. Passing parameters is similar to performing a series of assignment statements between parameter names and their argument values.
- If a function call raises no exception, these rules ensure that each parameter in the function header matches the value of exactly one argument in the function call. After Python binds each parameter name to its argument, it executes the body of the function, which computes and returns the result of calling the function.

Argument/Parameter Matching

- Here are some examples of functions that we can call to explore the rules for argument/parameter matching specified above. These functions just print their parameters, so we can see the arguments bound to them (or see which exception they raise).

```
def f(a,b,c=10,d=None): print(a,b,c,d)

def g(a=10,b=20,c=30) : print(a,b,c)

def h(a,*b,c=10)      : print(a,b,c)
```

Call	Parameter/Argument Binding (Matching Rule)
f(1, 2, 3, 4)	a=1, b=2, c=3, d=4(M1)
f(1, 2, 3)	a=1, b=2, c=3(M1); d=None(M4)
f(1, 2)	a=1, b=2(M1); c=10, d=None(M4)
f(1)	a=1(M1); c=10, d=None(M4); TypeError(M5c:b not matched)
f(1, 2, b=3)	a=1, b=2(M1); b=3(M3); c=10, d=None(M4) TypeError(M5b:b matched twice)
f(d=1, b=2)	d=1, b=2(M3); c=10(M4); TypeError(M5c:a not matched)
f(b=1, a=2)	b=1, a=2(M3); c=10, d=None(M4)
f(a=1, d=2, b=3)	a=1, d=2, b=3(M3); c=10(M4)
f(c=1, 2, 3)	c=1(M3); SyntaxError(M5a:2 is positional argument)
g()	a=10, b=20, c=30(M4)
g(b=1)	b=1(M3); a=10, c=30(M4)
g(a=1, 2, c=3)	a=1(M3); SyntaxError(M5a:2 is positional argument)
h(1, 2, 3, 4, 5)	a=1(M1); b=(2,3,4,5)(M2), c=10(M4)
h(1, 2, 3, 4, c=5)	a=1(M1); b=(2,3,4)(M2), c=5(M3)
h(a=1, 2, 3, 4, c=5)	a=1(M3); SyntaxError(M5a:2)
h(1, 2, 3, 4, c=5, a=1)	a=1(M1); b=(2,3,4)(M2); c=5(M3); TypeError(M5b:a matched twice)

Argument/Parameter Matching

- Here is a real but simple example of using *args, showing how the max function is implememented in Python; we dont' really need to write this function because it is in Python already, but here is how it is written in Python. We will cover raising exceptions later in this lecture note, so don't worry about that code.

```
def max(*args):# Can refer to args inside; it is a tuple of values
    if len(args) == 0:
        raise TypeError('max: expected >=1 arguments, got 0')
    answer = None
    for i in args:
        if answer == None or i > answer:
            answer = i
    return answer
print(max(3,-4, 2, 8)) # max with many arguments; prints 8
```

Argument/Parameter Matching

- In fact, the real max function in Python can take either (a) any number of arguments or (b) one iterable argument. It is a bit more subtle to write correctly to handle both parameter structures, but here is the code.

```
def max(*args) : # Can refer to args inside; it is a tuple of values
    if len(args) == 0:
        raise TypeError('max: expected >=1 arguments, got 0')
    if len(args) == 1: # Assume that if max has just one argument
        args = args[0] #   it's iterable, so take the max over its
value
    answer = None
    for i in args:
        if answer == None or i > answer:
            answer = i
    return answer
l = (3,-4, 2, 8)
print(max(l))           # max with one iterable argument; prints 8
```

Argument/Parameter Matching

- Finally, because of this approach computing max(3) raises an exception, because Python expects a single argument to be iterable. It might be reasonable in this case to return 3, but that is not the semantics (meaning) of the max function built into Python.

Argument/Parameter Matching

- Here is another real example of using *args, where I show how the print function is written in Python. The myprint calls a very simple version of print, just once at the end, to print only one string that it builds from args along with sep and end; it prints the same thing the normal print would print with the same arguments. Notice the use of the conditional if in the first line, to initialize s to either '' or the string value of the first argument.

Argument/Parameter Matching

```
def myprint(*args, sep=' ', end='\n'):
    s = (str(args[0]) if len(args) >= 1 else '')
    for a in args[1:]:              # all others come after sep
        s += sep + str(a)
    s += end   # end at the end
    print(s,end='') # print the entire string s
myprint('a',1,'x')  # prints a line
myprint('a',1,'x',sep='*',end='E') # prints a line but stays at end
myprint('a',1,'x')  # continues at end of previous line
```

- Together when executed, these print

```
a 1 x
a*1*xEa 1 x
```

Constructors

LECTURE 10

List, Tuples, Sets:

- Python's "for" loops allow us to iterate through all the components of any iterable data. We can even iterate through strings: iterating over their individual characters. Later in the quarter, we will study iterator protocols in detail, both the special iter/__iter__ and next/__next__ methods in classes, and generators (which are very very similar to functions, with a small but powerful twist). Both will improve our understanding of iterators and also allow us to write our own iterable data types (classes) easily.
- Certainly we know about using "for" loops and iterable data (as illustrated by lots of code above). What I want to illustrate here is how easy it is to create lists, tuples, and sets from anything that is iterable by using the list, tuple, and set constructors (we'll deal with dict constructors later in this section). For example, in each of the following the constructor for the list/tuple/set objects iterates over the string argument to get the 1-char strings that become the values in the list/tuple/set object.

List, Tuples, Sets:

```
l = list ('radar') then l is ['r','a','d','a','r']
t = tuple('radar') then t is ('r','a','d','a','r')
s = set  ('radar') then s is {'a','r','d'} or {'d','r','a'} or
...
```

- Note that lists/tuples are ORDERED, so whatever the iteration order of their iterator argument is, the values in the list/tuple will be the same order.
- Contrast this with sets, which have (no duplicates and) no special order. So, set('radar') can print its three different values in any order.

List, Tuples, Sets:

- Likewise, since tuples/sets are iterable, we can also compute a list from a list, a list from a a tuple, or a list from a set. Using l, t, and s from above.

```
list(t) which is ['r', 'a', 'd', 'a', 'r’]

list(s) which is ['r', 'd', 'a’]
   assuming s iterates in the order 'r', 'd', 'a’

list(l) which is ['r', 'a', 'd', 'a', 'r']
```

List, Tuples, Sets:

- The last of these iterates over the list to create a new list with the same values: note that l is list(l) is False, but l == list(l) is True: there are two different lists, but they store the same contents (see the next section on "is" vs "==" for more details).
- Likewise, we could create a tuple from a list/set, or a set from a list/tuple. All the constructors handle iterable data, producing a result of the specified type by iterating over their argument.

List, Tuples, Sets:

- Note that students sometimes try to use a list constructor to create a list with one value. They write list(1), by Python responds with "TypeError: 'int' object is not iterable" because the list constructor is expecting an iterable argument. The correct way to specify this list with one value is [1]. Likewise for tuples, sets, and dictionaries.
- Program #1 will give you lots of experience with these data types and when and how to use them. The take-away now is it is trivial to convert from one of these data types to another, because the constructors for their classes all allow iterable values as their arguments, and all these data types (and strings as well) are iterable (can be iterated over).

Dictionary Constructors:

- Before leaving this topic, we need to look at how dictionaries fit into the notion of iterable. There is not just ONE way to iterate through dictionaries, but there are actually THREE ways to iterate through dictionaries: by keys, by values, and by items (each item a 2-tuple with a key followed by its associated value). Each of these is summoned by a method name for dict, and the methods are named the same: keys, values, and items.
- So, if we write the following to bind d to a dict (we will discuss this "magic" constructor soon)

```
d = dict(a=1,b=2,c=3,d=4,e=5)

# the same as d = {'a':1,'b':2,'c':3,'d':4,'e':5}
```

Dictionary Constructors:

- Then we can create lists of three aspects of the dict:

```
list(d.keys  ()) is like ['c', 'b', 'a', 'e', 'd']
list(d.values()) is like [3, 2, 1, 5, 4]
list(d.items ()) is like [('c',3),('b',2),('a',1),('e',5),('d',4)]
```

- I said "is like" because sets and dicts are NOT ORDERED: in the first case we get a list of the keys; in the second a list of the values; in the third a list of item tuples, where each tuple contains one key and its associated value. But in all three cases, the list's values can appear in ANY ORDER.

Dictionary Constructors:

- Note that the keys in a dict are always unique, but there might be duplicates among the values: try the code above with d = dict(a=1,b=2,c=1). Items are unique because they contain keys (which are unique).
- Also note that if we iterate over a dict without specifying how, it is equivalent to specifyihg d.keys(). That is list(d) is the same as list(d.keys()) which is like ['a', 'c', 'b', 'e', 'd']

Dictionary Constructors:

- One way to construct a dict is to give it an iterable, where each value is either a 2-tuple or 2-list: a key followed by its associated value. So, we could have written any of the following to initialize d:

```
d = dict( [['a', 1], ['b', 2], ['c', 3], ['d', 4], ['e', 5]] ) #list of  2-list
d = dict( [('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)] ) #list of  2-tuple
d = dict( (['a', 1], ['b', 2], ['c', 3], ['d', 4], ['e', 5]) ) #tuple of 2-list
d = dict( (('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)) ) #tuple of 2-tuple
```

or, even (a tuple that has a mixture of 2-tuples and 2-lists in it)

```
d = dict( (('a', 1), ['b', 2], ('c', 3), ['d', 4], ('e', 5)) )
```

or even (a set of 2-tuples; we cannot have a set of 2-list (see hashable below)

```
d = dict( {('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)} )
```

Dictionary Constructors:

- The bottom line is that the dict argument must be iterable, and each value in the iterable must have 2 values (e.g., a 2-list or 2-tuple) that represent a key followed by its associated value.
- Finally, if we wanted to construct a dict using the keys/values in another dict, here are two easy ways to do it

```
d_copy = dict(d))
```

or

```
d_copy = dict(d.items())
```

Sharing/Copying: is vs. ==

LECTURE 11

Sharing/Copying: is vs. ==

- It is important to understand the fundamental difference between two names sharing an object (bound to the same object) and two names referring/bound to "copies of the same object". Note that if we mutate a shared object, both names "see" the change: both are bound to the same object which has mutated. But if they refer to different copies of an object, only one name "sees" the change.
- Note the difference between the Python operators is and ==. Both return boolean values. The first asks whether two references/binding are to the same object (the is operator is called the object-identity operator); the second asks only whether the two objects store the same values. See the different results produced for the example below. Also note that if x is y is True, then x == y must be True too: an object ALWAY stores the same values as itself. But if x == y is True, x is y may or may not be True.

Sharing/Copying: is vs. ==

- For example, compare execution of the following scripts: the only difference is the second statement in each: y = x vs. y = list(x)

```
x = ['a']
y = x        # Critical: y and x share the same reference
print('x:',x,'y:',y,'x is y:',x is y,'x == y:',x==y)
x [0] = 'z' # Mutate x (could also append something to it)
print('x:',x,'y:',y,'x is y:',x is y,'x == y:',x==y)
```

- This prints

```
x: ['a'] y: ['a'] x is y: True x == y: True
x: ['z'] y: ['z'] x is y: True x == y: True
```

Sharing/Copying: is vs. ==

```
x = ['a']
y = list(x)
# Critical: y refers to a new list with the same contents as x
print('x:',x,'y:',y,'x is y:',x is y,'x == y:',x==y)
x [0] = 'z' # Mutate x (could also append something to it: x+)
print('x:',x,'y:',y,'x is y:',x is y,'x == y:',x==y)
```

This prints

```
x: ['a'] y: ['a'] x is y: False x == y: True
x: ['z'] y: ['a'] x is y: False x == y: False
```

Sharing/Copying: is vs. ==

- You might have learned about the id function: it returns a unique integer for any object. It is often implemented to return the first address in memory at which an object is stored, but there is no requirement that it returns this int.
- Checking "a is b" is equivalent to checking "id(a) == id(b)" but the first way to do this check is preferred.
- Finally there is a copy module in Python that defines a copy function: it copies some iterable without us having to specify the specific constructor (like list, set, tuple, or dict).

Sharing/Copying: is vs. ==

- So, we can import it as: from copy import copy
- Assuming x is a list, we can replace y = list(x) by y = copy(x).
- Likewise, if x is a dict we can replace y = dict(x) by y = copy(x).

Sharing/Copying: is vs. ==

- We could also just: import copy (the module) and then write y = copy.copy(x) but it is clearer in this case to write "from copy import copy". Here is a simple implementation of copy:

```
def copy(x)
    return type(x)(x)
```

which finds the type of x, then constructs a new object of that type, initialized by x.

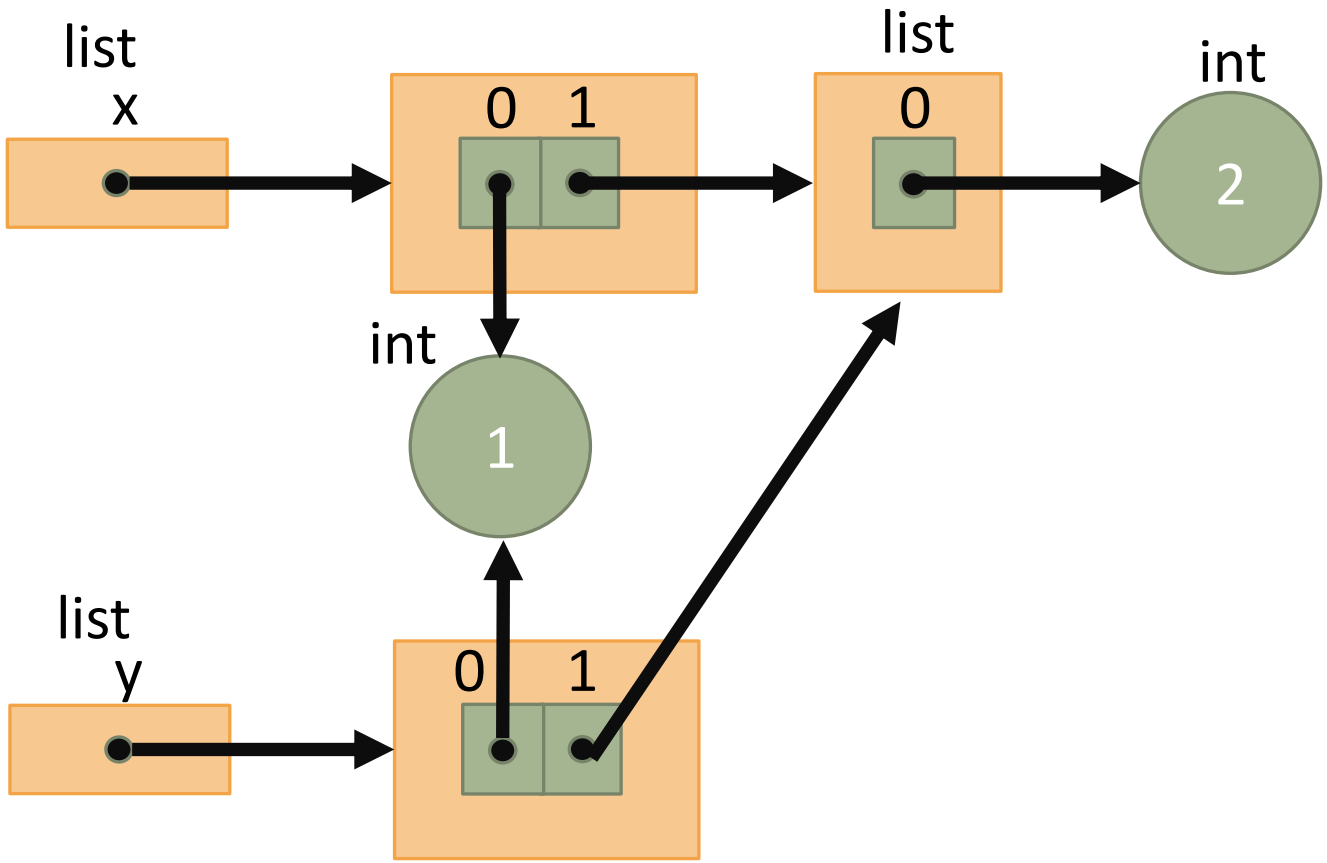
Sharing/Copying: is vs. ==

- Note that copying in all the ways that we have discussed is SHALLOW. That means the "copy" is a new object, but that object stores all the references from the "object being copied". For example, if we write

```
x = [1, [2]]
y = list(x) # or y = copy(x) or y = x[:]
```

then we would draw the following picture for these assignments.

Sharing/Copying: is vs. ==

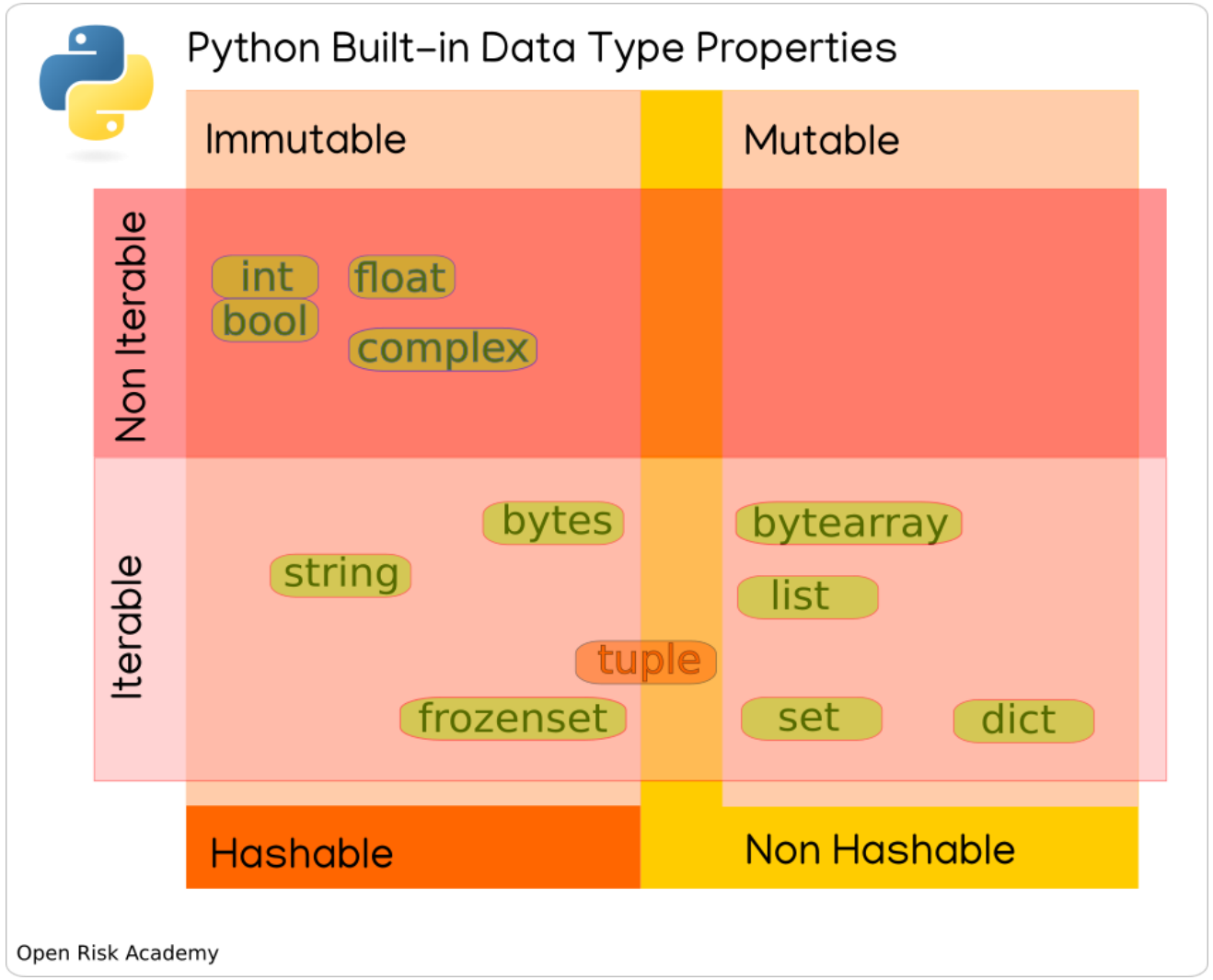


Sharing/Copying: is vs. ==

- Here, x and y refer to DIFFERENT lists: but all the references in x's list are identical to the references in y's list. Using the 'is" operator we can state that (1) x[0] is y[0] == True, and (2) x[1] is y[1] == True.
- This means if we now write x[1][0] = 'a', then print(y[1][0]) prints 'a' too, because x[1] and y[1] refer to the same list.
- We will discuss how to do DEEP copying when we discuss recursion later in the course; it is implemented in the copy module by the function deepcopy, which does a deep copy of an object, by constucting an object of that type and populating it with deep copies of all the objects in the original object.

Hashable vs. Mutable and how to Change Things

LECTURE 12


Python Built-in Data Type Properties
Immutable
Mutable
Non Iterable
int
float
bool
complex
Iterable
bytes
bytearray
string
list
tuple
frozenset
set
dict
Hashable
Non Hashable
Open Risk Academy

Hashable vs. Mutable:

how to Change Things

- Python uses the term Hashable, which has the same meaning as Immutable. So hashable and mutable are OPPOSITES: You might see this message relating to errors when using sets with UNHASHABLE values or dicts with UNHASHABLE keys: since hashable means immutable, then un-hashable means un-immutable which simplifies (the two negatives cancel) to mutable. So unhashable is mutable. So

```
hashable    is immutable
unhashalble is mutable
```

- Here is a quick breakdown of standard Python types

```
Hashable/immutable: numeric values, strings, tuples containing
                    hashable/immutable data, frozenset
mutable/unhashable: list, sets, dict
```

Hashable vs. Mutable:

how to Change Things

- The major difference between tuples and lists in Python is the former is (mostly) hashable/immutable and the later is not. I say mostly because if a tuple contains mutable data, you can mutate the mutable data. So

```
x = (1, 2, [3, 4])
x[2][0] = 'a'
print(x)
```

prints

```
(1, 2, ['a', 4])
```

Hashable vs. Mutable:

how to Change Things

- So, you have mutated the tuple. On the other hand, you cannot append to a tuple, nor store a new value in one of its indexes (x[0] = 10 is not allowed), nor sort it.
- So technically, a tuple storing hashable/immutable values is hashable/immutable, but a tuple storing unhashable/mutable values is unhashable/mutable. So, if some other datatype (e.g., values in a set, or keys in a dictionary) needs to be hashable/immutable, use a tuple (storing hashable/immutable values) to represent its value, not a list. Thus, we cannot say, "Any value whose type is a tuple is hashable/immutable."
- Instead, we must say, "A value whose type is a tuple is hashability/immutability when the tuple stores only hashable/immutable values."

Hashable vs. Mutable:

how to Change Things

- A frozenset can do everything that a set can do, but doesn't allow any mutator methods to be called (so we cannot add a value to or delete a value from a frozenset). Thus, we can use a frozen set as a value in a set or a key in a dictionary.
- The constructor for a **frozenset** is **frozenset(...)** not **{}**. Note that once you've constructed a frozen set you cannot change it (because it is immutable). If you have a set s and need an equivalent **frozenset**, just write **frozenset(s)**.

Hashable vs. Mutable:

how to Change Things

- The function hash takes an argument that is hashable (otherwise it raises TypeError, with a message about a value from an unhashable type) and returns an int.
- We will study hashing towards the end of the quarter: it is a technique for allowing very efficient operations on **sets** and **dicts**. ICS-46 (Data Structures) studies hash tables in much more depth, in which you will implement the equivalent of Python **sets** and **dicts** by using hash tables.