CIS 41B
Advanced Python Programming

Iterables

De Anza College
Instructor: Clare Nguyen

Review Basic Data Types

- Integer types:
  - int: integers or whole numbers
  - bool: Boolean values True or False
- Floating point types:
  - float: floating point numbers
  - complex: real and imaginary floating point numbers
  - decimal: more precise floating point representation, need to import decimal module
- String type
  - str: a sequence of 1 or more characters, need single or double quotes
  - a character is a string of 1 element
  - str methods
- Common type conversion:
  - int( ): convert a string to an int, exception if string can't be converted
  - float( ): convert a string to a float, exception if string can't be converted
  - str( ): convert a number to a string
  - ord( ): convert a character to its Unicode number
  - chr( ): convert a Unicode number to a character

© 2019 C. Nguyen

Checking Data Type

- To check for data type, Python provides the isinstance function:

```
isinstance(variable, type)     # returns True or False
```

- This function can be used to check whether an object is part of a specific inheritance hierarchy, such as:

```
isinstance(myVehicle, Truck)        # where Truck is a subclass in a hierarchy
```

- For general purpose data type checking, however, it is more Pythonic to simply assume that the data is a particular type and see if it works.
- Example of converting an input string var to a number. If the string can't be converted, print an error message.

Non-Pythonic type checking:

```
if isinstance(var, int) :
    var = int(var)
else :
    print("var is not a number")
```

Pythonic type checking:

```
try :
    var = int(var)
except ValueError :
    print("var is not a number")
```

- The example above demonstrates the concept of duck typing in Python. Duck typing is not unique to Python, it means to assume that a particular object is the correct type and use it as such. If it works, then it must be the correct type. There's no need for type checking.

© 2019 C. Nguyen

Review Iterable

- A data type that can be iterated over (or stepped through one element at a time) is called an iterable.
- An iterable supports the in operator to check for existence of an element, and the len function to get the number of elements.
- Since an iterable contains multiple data values, it's also called a container.
- Common iterable operations
- Some common built-in iterables:
  - list: a mutable, ordered sequence of data of any type (list methods)
  - str: an immutable, ordered sequence of characters
  - tuple: an immutable, ordered sequence of data of any type
  - set: a mutable, unordered sequence of unique data (set methods). The data type in a set must be immutable.
  - dictionary: a mutable, unordered sequence of key-value pairs, where the key must be unique and immutable, but the value can be any data type (dictionary methods)
- The ordered sequences (list, str, tuple) support indexing with the [ ] operator and slices [n:m]
- The dictionary also supports the [ ] operator but inside the [ ] is a key and not an index.

© 2019 C. Nguyen

Comprehension

- Comprehension is supported by iterables that are mutable
  - List comprehension:

```
[ expression  for item in iterable  if condition ]
```

  - Set comprehension:

```
{ expression  for item in iterable  if condition }
```

  - Dictionary comprehension:

```
{ key_expression : value_expression  for key,value in iterable  if condition }
```

```
{ key_expression : value_expression  for key in iterable  if condition }
```

- Comprehension should be used when an iterable is being created by adding one data element at a time to the iterable.

© 2019 C. Nguyen

Comprehension for 2D Structures

- When we need to create a two-dimensional containers by adding one data item at a time, it is possible to use comprehension.
- The following is a 2D comprehension to create a *list* from a table:

```
[expression(item)  for row in table  for item in row]
```

- The following is a 2D comprehension to create a *table* from another table:

```
[ [expression(item) for item in row]  for row in table]
```

- Example: create a table of positive numbers from an existing table

```
table = [ [1,-3,5,6], [2,3,-1,9], [2,8,-4,-8] ]
positives = [ [n for n in row if n>0] for row in table]
print(positives)
# output:  [ [1, 5, 6], [2, 3, 9], [2, 8] ]
```

© 2019 C. Nguyen

Collections: Default Dictionary (1)

- When accessing a dictionary with a non-existing key, a KeyError exception is produced.
- If we're not sure that the key exists, we need to use the get method:

```
aValue = myDict.get(aKey, defaultValue)
```

- A default dictionary behaves just like a regular dictionary, but we can access the default dictionary with a non-existing key and not get an exception.
- The default dictionary will automatically fill in a default value if the key doesn't exist.
- A default dictionary is part of the collections module, which we need to import.
- Format:

```
myDefaultDict = collections.defaultdict(default_data_type)
```

  where default_data_type is the data type of the default value.

- For

data type	The default value is
int	0
float	0.0
sequence	empty sequence

© 2019 C. Nguyen

Collections: Default Dictionary (2)

- Example:

1. Define the default dictionary

```
letterCount = collections.defaultdict(int)
```

2. Use it to count letters

```
for char in myStr :
    letterCount[char] += 1

    # The above is more intuitive than using a regular dictionary and
    # using:
    # letterCount[char] = letterCount.get(char, 0) + 1
```

© 2019 C. Nguyen

Iterable Operators

- The following table shows common iterable operators.

Operator	Explanation	Example
+	Concatenate 2 iterables into a new iterable	list1 + list2
* (binary)	Replicate an iterable n times	set1 * 3
in	Return True if an item is in the iterable; False otherwise	item in list1
* (unary, LHS)	Packing: group all available items into an iterable	first, *rest = [1,2,3,4] # first is 1, rest is [2, 3, 4]
* (unary, RHS)	Unpacking: split an iterable into multiple items, used in argument passing	tuple1 = (2, 3, 4) def function(a, b, c) : pass # call: function(*tuple1) # same as: function(2, 3, 4)

- Out of the common iterable operators above, the packing and unpacking operators are more advanced use of iterables and are covered in the next slides.

© 2019 C. Nguyen

The Unpacking Operator *

- In an assignment statement: `dest_var = source_var`
  - The destination variable on the left side of the = operator receives data, which means it has LHS or left hand side context.
  - The source variable on the right side of the = operator provides the data, therefore the source variable has RHS or right hand side context.
- When we use the * operator:
  1. In front of a sequence data type (such as a list or tuple)
  2. And the sequence is used in RHS context (ie. it provides the data)
  3. And the LHS requires multiple individual data values.

  Then Python will interpret the * as the unpacking operator.
- The unpacking operator separates or unpacks a sequence of data into individual data values.
- Example of using the unpacking operator with a list `L = [8, 2, 5]`

```
print(L)          # output:  [8, 2, 5]     print a list of integers
print(*L)         # output:  8  2  5       print 3 individual integers
print(8, 2, 5)    # output:  8  2  5       same as the unpacked L
```

- The unpacking operator is often used in argument passing, to unpack an iterable into individual parameters as the function requires.

© 2019 C. Nguyen

The Packing Operator *

- Keeping in mind the LHS and RHS context, when we use the * operator:
  1. In front of a sequence data type (such as a list or tuple)
  2. And the sequence is being used in LHS context (ie. it receives data)
  3. And the RHS is made of multiple individual data values

  Then Python will interpret the * as the packing operator.
- The packing operator groups together or packs individual data values into a sequence.
- Example of using the packing operator to divide a list

```
L = [8, 3, 0, 2, 7]
(var1, var2, *theRest) = L      # var1 is 8, var2 is 3, and
                                # theRest is [0, 2, 7]
(var1, var2, theRest) = L       # Error!
                                # Without the packing operator
                                # there are too few LHS variables
```

© 2019 C. Nguyen

Iterable Functions (1)

- The following table shows common iterable functions, in alphabetical order.

Function	Explanation	Example
all()	Return True if every element evaluates to True or if the iterable is empty. Return False otherwise	all(elem % 2 for elem in iterable)
any()	Return True if one element evaluates to True; Return False otherwise or if the iterable is empty.	any(elem > 0 for elem in iterable)
enumerate()	Used with for in to get a count and the element of an iterable	for count, elem in enumerate(iterable, start=n) # count starts at n and counts up
len()	Return the number of elements of the iterable	len(iterable)
max() min()	Return the max or the min value of the iterable	max(iterable) min(iterable)

© 2019 C. Nguyen

Iterable Functions (2)

Function	Explanation	Example
reversed()	Return an iterator in reversed order	for elem in reversed(iterable) # iterate from the end to the front of iterable # to fetch elem
sorted()	Return an iterable in sorted order	sorted(iterable) # ascending sorted(iterable, reverse=True) # descend.
sum()	For an iterable of numbers: return the total of all element values	sum(iterable, n) # return: sum of all elements + n # n defaults to 0
zip()	Return an iterator of tuples	for t in zip(seq1, seq2, seq3) # t is a tuple of corresponding elements # from seq1, seq2, and seq3 # number of iteration = shortest list length

The len(), max(), min(), reversed(), sorted(), sum() functions are self-explanatory. The other functions' examples are in the next slides.

© 2019 C. Nguyen

Example: all() and any()

```
L = [8, 4, 3, -2, 0, 2, 11, -7, 6]
T = tuple()

if any(elem < 0 for elem in L) :                # True
    print("Negative value detected in L")

if any(elem > 100 for elem in L) :              # False
    print("At least 1 value larger than 100")

if all(elem > 0 for elem in L) :                # False
    print("All positive values")

if all(elem != 0 for elem in T) :               # True
     print("All non-zero in T")


#  Screen output:
#  Negative value detected in L
#  All non-zero in T
```

© 2019 C. Nguyen

Example: enumerate()

```
T = ('A', 'B', C')
for num, letter in enumerate(T, 65) :
      print(num, letter)
print()
for num, letter in enumerate(T) :
      print(num, letter)
```

Screen output:

```
65  A
66  B
67  C

0  A
1  B
2  C
```

© 2019 C. Nguyen

Example: zip()

```
L1 = (1, 3, 5, 7, 9)
L2 = (2, 4, 6, 8)
L3 = (100, 200, 300, 400, 500)

for elem in zip(L1, L3) :
    print(elem)

D = dict(zip(L3, L1))
print(D)

D = dict(zip(L1, L2, L3))

for elem in zip(L1, L2, L3) :
    print(elem)
```

Screen output:

```
(1, 100)
(3, 200)
(5, 300)
(7, 400)
(9, 500)

{100: 1, 200: 3, 300: 5, 400: 7, 500: 9}

Error! Only 2 iterables allowed

(1, 2, 100)
(3, 4, 200)
(5, 6, 300)
(7, 8, 400)     # only 4 iterations
```

© 2019 C. Nguyen

Copying Iterables

- Recall that with iterables, the following code does not create a new list L2 which is a copy of the list L1:

```
L1 = [8, 2, 5]
L2 = L1          # L2 is an alias or another name for the list at L1
```

- With an iterable, we must do an actual copy by using a method to create a new iterable object. Two common ways to copy a 1D list:

```
L2 = list(L1)        # use tuple(), set(), dict() in a similar way
# or:
L2 = L1.copy()     # now L2 is a list that is separate but identical to L1
```

- But for a 2D or higher dimensional iterable (list of tuples or tuples of dictionaries of lists), we need to do a deep copy.
- Deep copying means we must make a new object of each internal iterable.

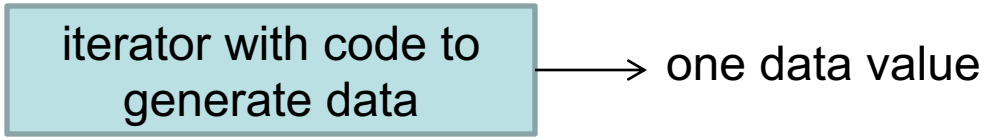
```
import copy
# if list1 is a list of 4 internal lists, a deep copy will copy 5 lists:
#  -  list1, which is a list of references
#  -  the 4 internal lists that are referenced by list1
list2 = copy.deepcopy(list1)     # list2 is separate but identical to list1
```

© 2019 C. Nguyen

Iterable and Iterator

- There are 2 terms, iterable and iterator, that sound similar but have different meaning.
- An iterable is an object that:
  - contains *multiple* data values that are stored and accessed in a specific way
  - has multiple methods to work with the data
  - the user can iterate or walk through the data values one at a time
- An iterator is an object that:
  - contains *one* data value that is part of a sequence of data
  - has a method to generate and return the next data value in the sequence
- An iterator can work with an iterable, as a "front end" to the iterable.
  - In this case the iterator uses the sequence of data from the iterable and returns one data value at a time

- An iterator can also be a stand alone object.
  - In this case it has code to generate one data at a time from an algorithm



© 2019 C. Nguyen

Iterator That Works With An Iterable (1)

- We have been using iterators indirectly in some of our code.
- An iterator is actually the mechanism used "under the hood" in a for loop. In this for loop:

```
for elem in tupleA :
    print(elem)
```

The actual code that Python runs is:

```
i = iter(tupleA)                  # create iterator which is the  "front end" for tupleA
while True :
    try :
        print(next(i))            # use next to get one data values from the iterable
    except StopIteration :        # end loop when StopIteration exception occurs
        break
```

- To use an iterator i that works with an iterable, as shown above:
  - Create an object of the iter class and pass in the iterable. The iterator now becomes the "front end" to the iterable.
  - Write a while loop that stops (breaks) when a StopIteration exception is detected. StopIteration occurs when there is no more data in the iterable.
  - In each iteration of the while loop, call next() to get the next data in the sequence.

© 2019 C. Nguyen

Iterator That Works With An Iterable (2)

- Iterators are also used "under the hood" to implement functions that work with iterables: range(), enumerate(), and zip(). All are functions that allow us to fetch one data at a time from the iterable.
- If we want to print the entire data sequence from these functions, such as zip(), we cannot simply print from zip():

```
L1 = [1, 3, 5]
L2 = [2, 4, 6]
print(zip(L1, L2))          # output:  <zip object at 0x0000001E29079120>
                            # the address of the iterator object is printed,
                            # not the sequence of tuples
```

- We can use a for loop so that the next() function will be called:

```
for elem in zip(L1, L2) :
    print(elem, end=" ")    # output:  (1, 2) (3, 4) (5, 6)
```

- Or, taking advantage of the fact that an iterator will produce the sequence of data, we can use the unpacking operator:

```
print(*zip(L1, L2))         # output:  (1, 2) (3, 4) (5, 6)
                            # print or LHS accepts multiple values, and zip or RHS
                            # has a sequence of data
```

© 2019 C. Nguyen

Stand Alone Iterator

- An iterator doesn't have to work with an iterable in order to have the data sequence. It can also be implemented such that it generates its own data sequence.
- This is where the main advantage of iterators come through:
  - When an iterator generates its own data sequence and returns one data at a time, it is extremely memory efficient.
  - Because it only generates one data value at a time, on demand, it doesn't take up the large amount of memory to store an entire sequence.
  - This means that an iterator can work with an infinite sequence of data!
- Example of the advantage of iterators:
  In an application we need to work with up to the first 1 million prime numbers. We have 2 choices:
  - We can write a loop that generates a million primes and store them in a list. This list will take up a lot of memory due to its size.
  - We can implement an iterator that generates one prime number at a time. Then we call the iterator's next() whenever we need the next prime number. There's no huge list that takes up memory.

© 2019 C. Nguyen

Writing an Iterator (1)

- We can write our own iterator when we need to generate values for a large sequence of data.
- The iterator needs to be a class that has these 2 required methods:

  __iter__() this method makes the class an iterable by returning itself

  __next__() this method makes the class an iterator by returning the next value in the sequence

- Example:

  We want to continue to produce the powers of 2 (such as: 1, 2, 4, 8, 16…) as long as the user wants to continue.

- Issues:
  1. The powers of 2 sequence is infinite
  2. We only want the first N values, where N is determined by the user. The user may want just 3 values or the user could want 1000 values.
  3. The user only needs one value at a time, we don't need to store all the values.
- Solution:
  - Create a Powers2 class which is an iterator class
  - Instantiate the Powers2 iterator from the class
  - Call next() with the Powers2 iterator to get a powers of 2 value

© 2019 C. Nguyen

Writing an Iterator (2)

- Implementation:

```
class Powers2 :
    def __init__(self) :
        self._exp = 0                # start at 2^0
    def __iter__(self) :             # required method
        return self
    def __next__(self) :             # required method
        value, self._exp = self._exp, self._exp + 1    # save current exponent, and
                                                       # increment exp for next time
        return 2**value              # return 2^current exponent
```

- Note that only one data value is generated by the iterator. There is no memory used to store multiple powers of 2.
- Using the iterator:

```
p2 = Powers2()                   # create iterator
print(next(p2))                  # output:  1
for i in range(3) :
    print(next(p2))              # output:  2
                                           4
                                           8
```

© 2019 C. Nguyen

Generator

- A generator is a simple way to implement an iterator class.
- Most of the time when we want an iterator, we write a generator instead, and Python will create an iterator “under the hood” for us, similar to how Python creates an iterator when we write a for loop.
- There are two ways to write a generator:
  - A generator *expression*: simplest coding but it can only be used with a finite sequence of data.
  - A generator *function*: more coding (not as much as writing an iterator directly) but it can work with an infinite sequence.

© 2019 C. Nguyen

Generator Expression

- A generator expression looks similar to a comprehension, except we use ( )
- Format:

```
gen = (elem  for elem in iterable  if condition)
```

- Usage:

```
val = next(gen)      # get next value in iterable that matches condition
```

- Example of a generator for powers of 2:

```
p2 = (2** num for num in range(1, 1001))     # go up to 1000 for exponent
print(next(p2))                  # print the first power of 2
for i in range(3) :
    print(next(p2))              # print the next 3 powers of 2
```

© 2019 C. Nguyen

Generator Expression vs. Comprehension

- In the previous slide we used a generator expression:

```
p2 = (2** num for num in range(1, 1001))
```

- We could instead use a comprehension:

```
p2 = [2** num for num in range(1, 1001)]
```

- Both can be used to get multiple powers of 2. What's the difference between them, other than the ( ) or [ ]?
- Comprehension expression:
  - *Pro*: Because the list is stored, we can walk forward or backward to fetch any data as many times as we like.
  - *Con*: Produces a list with all 1000 data values, which needs to be stored and takes up memory space. And we might not need all 1000 values.
- Generator expression:
  - *Pro*: Produces no data value unless requested with next(), so no data storage needed. This on-demand data generation is also called lazy evaluation, the work to generate data is only done when needed.
  - *Con*: Can only goes to the next data (no going back to previous data), and once data is fetched with next(), we cannot fetch it again.

© 2019 C. Nguyen

Generator Function

- When we want a generator that works with an infinite sequence of data, we need to write a generator function.
- From the generator function Python will create an iterator for us, just like how Python creates an iterator from a generator expression.
- The generator function must include a yield keyword, which returns the next value in the sequence. It is the yield that makes Python interpret the function as a generator function.
- Example of a powers of 2 generator, note how it's simpler than the iterator that we saw in a previous slide:

```
def Powers2() :
    exp = 0
    while True :
        yield 2 ** exp        # causes Python to create an iterator from this function
        exp += 1
```

- Using the generator exactly the same way as with the iterator previously:

```
p2 = Powers2()                # create iterator
print(next(p2))               # output:  1
for i in range(3) :
    print(next(p2))           # output:  2
                                       4
                                       8
```

© 2019 C. Nguyen

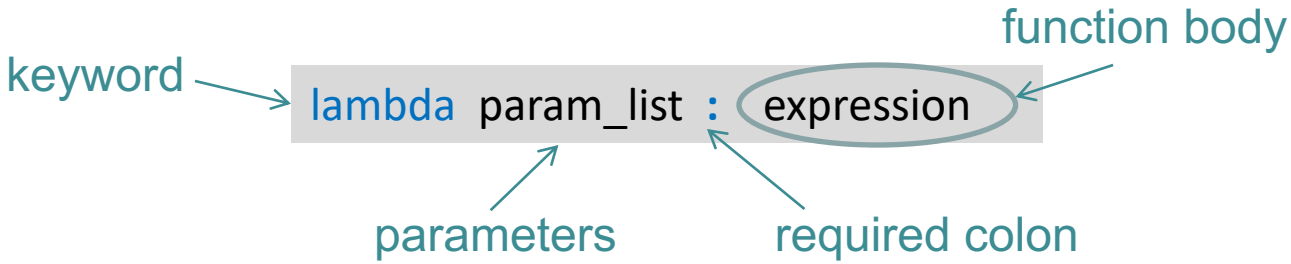
Advantages of Iterators and Generators

- Iterators and generators are very efficient ways to get a subset of consecutive values from a large data sequence.
- When we want to get a few powers of 2 values, both the iterator and the generator create the next power of 2 value on demand.
- This eliminates the CPU time to generate extra values that may not be needed if the user only wants a couple of values.
- This also saves a lot of memory space because there's no need to keep a big list of powers of 2 around in case the user wants many values.
- A generator is a shortcut to writing a full iterator. We write the generator either in the format of a function with yield or as a generator expression, and let Python do the "heavy lifting" of creating an iterator for us.
  For this reason, generators are written more often than iterators.
- This doesn't mean that there's no need to write an iterator. A generator only moves forward (move to the next element in the sequence). If we want to be able to "undo" the previous next() call, we can write the iterator and provide a backup() method to go back to the previous element.

© 2019 C. Nguyen

Lambda Expression (1)

- A lambda expression is a simple way to create an anonymous function (a function without a name), where the function body is short.
- Anonymous functions are typically called in one place in the code and don't need to be kept around like a named function would.
- Format:



  - Instead of a function name, the keyword lambda is used
  - param_list is a comma separated list of input parameters (just like with a named function). Example: num1, num2
  - The colon is required, it separates the parameters and the function body
  - The body of the function is an expression which will be evaluated to get a resulting value, and the value will be returned.

© 2019 C. Nguyen

Lambda Expression (2)

- Example: write a lambda function to add 2 and 3, print the result

```
print( (lambda x, y : x + y) (2, 3) )
```

- The above lambda expression has:
  - 2 parameters x and y
  - A function body which is the expression x + y
- Since the lambda expression is an anonymous function, the input argument list is put immediately after the function, as with any function.
  In this case the argument list is (2, 3).
- When the code runs:
  - 2 and 3 are passed to the lambda function and get stored in x and y
  - In the function body, 2 + 3 evaluates to 5, and 5 is returned
  - 5 becomes the input argument to print(), which prints 5 to screen
- Lambda expressions are typically used as an input argument to another function, where that function requires a function input argument, as shown in the next slides.

© 2019 C. Nguyen

map Function

- The map function takes 2 input arguments: a function name and an iterable.
- map will apply the input function to each element of the input iterable and return an iterator, which can be converted to an iterable.
- Format: `map(aFunction, anIterable)`
- Example:

```
origList = [1, 2, 3, 4]
def add1(n) :
    return n+1
newList = list(map(add1, origList))                 # using a named function
newList = list(map(lambda n: n+1, origList))        # using a lambda expression,
                                                    # no need to define add1()
# Either way, map will take each element of origList and run the input
# function with it. Then we convert all the resulting values into a newList.
# The resulting newList is:  [2, 3, 4, 5]
```

- What's a shorter way to add 1 to each value of origList and create newList?

© 2019 C. Nguyen

filter Function

- The filter function takes 2 input arguments: a function name and an iterable.
- The input function must return True or False.
- filter applies the input function to each element of the input iterable, and only elements that evaluate to True will be returned as part of an iterator, which can be converted to an iterable.
- Format:

```
filter(boolFunction, anIterable)
```

- Example:

```
origTuple = (1, 10, 2, 30, 5, 7, 45)
newTuple = tuple(filter(lambda n: n>=10, origTuple)))
# newTuple is  (10, 30, 45)
```

- What's a shorter way to filter values that are above 10 and create a newTuple?
- Generally most Python programmers prefer to use comprehension and generator instead of map and filter. Comprehension and generator are considered more readable and don't require a lambda expression. map and filter are more likely used by those doing functional programming.

© 2019 C. Nguyen

reduce Function

- The reduce function is typically discussed together with map and filter since all 3 functions work with an interable.
- However, reduce is not part of the Python core and must be imported with:

  from functools import reduce
- reduce takes 2 input arguments: a function name and an iterable.
- The input function takes 2 input arguments: the 1st and 2nd elements of the iterable. When the function runs, the 2 input are effectively removed from the iterable and reduced to 1 return value, which replaces the 1st element of the iterable.
- Format:

```
reduce(functionWith2InputArgs, anIterable)
```

- Example:

```
L = [1, 2, 3, 4, 5]
result = reduce(lambda x,y: x + y, L)      # result is 15
```

- Step by step result of reduce example above:

1. iterable:	1 2 3 4 5	reduce: 1 + 2 => 3
2. iterable:	3 3 4 5	reduce: 3 + 3 => 6
3. iterable:	6 4 5	reduce: 6 + 4 => 10
4. iterable:	10 5	reduce: 10 + 5 => 15

© 2019 C. Nguyen

sorted Function

- The sorted function takes an iterable as input and returns an ordered iterable that is sorted in ascending or descending order, depending on the reverse input argument.

```
newIterable = sorted(iterable, reverse=True)
```

- If the data to be sorted is composite data (it contains multiple fields or is an object with multiple attributes), we can tell sorted to sort by a specific field or attribute by using the key input argument:

```
sorted(iterable, key=function)
```

  where the key function takes one input argument (the data to be sorted) and returns a value in that data that is used for sorting.

- Example 1:

```
L1 = [ (5, 19, 23), (27, 31, 12), (9, 25, 17), (14, 10, 23) ]
print( sorted(L1, key=lambda t : t[1]) )      # sort by 2nd element
# output: [(14, 10, 23), (5, 19, 23), (9, 25, 17), (27, 31, 12)]
```

- Example 2:

```
# if data is a list of Student objects, each with a name attribute
print( sorted(data, key=lambda s : s.getName) )      # sort by name
```

© 2019 C. Nguyen

Up Next: Callables

© 2019 C. Nguyen