Lesson 6: Lists, Dictionaries, and Tuples

Data Structures

Data structures are basically just that - they are structures which can hold some data together. In other words, they are used to store a collection of related data.

There are four built-in data structures in Python - list, tuple, dictionary and set. We will see how to use each of them and how they make life easier for us.

List

A **list** is a data structure that holds an ordered collection of items i.e. you can store a sequence of items in a list. Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type.

This is easy to imagine if you can think of a shopping list where you have a list of items to buy, except that you probably have each item on a separate line in your shopping list whereas in Python you put commas in between them.

```
# This is my shopping list
shoplist = ['apple', 'mango', 'carrot', 'banana']
```

The list of items should be enclosed in square brackets `[` and `]` so that Python understands that you are specifying a list. Once you have created a list, you can add, remove or search for items in the list. Since we can add and remove items, we say that a list is a *mutable data type* - in other words this type can be modified.

The values in list are called ***elements*** or sometimes ***items***.

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

```
list_1 = [10, 20, 30, 40]
list_2 = ['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example (`list_1`) is a list of four integers. The second (`list_2`) is a list of three strings.

Go online to complete the
Create a List Challenge

Another way to create a new list is to use the built-in `list()` function:

Code	Output
`list_3 = list()` `print(list_3)`	`[]`

The `list` function without arguments will create an empty list. You can also pass in a single object to be converted into a list:

Code	Output

`list_4 = list(range(3))` **`print(list_4)`**	`[0, 1, 2]`

Go online to complete the Using list() and Range to Build a List Challenge

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

Code	Output
`cheeses = ['Cheddar', 'Edam', 'Gouda']` **`print(cheeses[0])`**	`Cheddar`

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

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is ***nested***.

When you want to access an element within a nested list, you need to use multiple indices. Look at this example:

```
my_bills = [["electric", 50], ["gas", 60], ["cell", 60]]
```

`my_bills[0]` references the first element in the list. The first element is another list `["electric", 50]`.

To get the name of the individual bill (`"electric"`) we need the first element of the first element of the `my_bills` list. We reference it like this:

Code	Output
`my_bills[0][0]`	`electric`

To get the dollar amount, we need the second element of the first element of the `my_bills` list:

Code	Output
`my_bills[0][1]`	`50`

Go online to complete the 2D Lists Challenge

A list that contains no elements is called an ***empty list***; you can create one with just a pair of empty brackets, `[]`.

As you might expect, you can assign list values to variables:

Code	Output

`cheeses = ['Cheddar', 'Edam', 'Gouda']`
`numbers = [17, 123]`
`empty = []`
 	
`print(cheeses, numbers, empty)`
	`['Cheddar', 'Edam', 'Gouda'] [17, 123] []`

Lists are Mutable

Unlike strings, lists are ***mutable*** because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

Code	Output

`numbers = [17, 123]`
 	
`numbers[1] = 5`
 	
`print(numbers)`
	`[17, 5]`

The one-eth element of numbers, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This relationship is called a mapping; each index "maps to" one of the elements.

Index	Map
`0`	17
`1`	5

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an IndexError.
- If an index has a negative value, it counts backward from the end of the list.

Go online to complete the Changing Lists Challenge

Lists and Functions

There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops:

Given:

```
numbers = [3, 41, 12, 9, 74, 15]
```

Function	Description	Example Code	Example Output
len()	Return the length of a list	`len(numbers)`	`6`
max()	Return the maximum value in the list	`max(numbers)`	`74`
min()	Return the smallest value in the list	`min(numbers)`	`3`
sum()	Return the sum of all the elements in a list	`sum(numbers)`	`154`

The `sum` function only works when the list elements are numbers. The other functions (`len`, `max`, and `min`) work with lists of strings and other types that can be comparable. The `min` and `max` functions look at the ASCII values of the characters which make up the string – just like with the comparison operators.

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
```

Code	Result
len(cheeses)	`3`
max(cheeses)	`Gouda`
min(cheeses)	`Cheddar`
sum(cheeses)	`TypeError: unsupported operand type(s) for +: 'int' and 'str'`

Go online to complete the The length of a list Challenge

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
my_list = ['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

Our elements are:

Index	Code
0	`'spam'`
1	`1`
2	`['Brie', 'Roquefort', 'Pol le Veq']`
3	`[1, 2, 3]`

Traversing a List

The most common way to traverse the elements of a list is with a `for` loop. The syntax is the same as for strings:

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
```

Code	Return

`for cheese in cheeses:`
`    print(cheese)`
	`Cheddar`
	`Edam`
	`Gouda`

Go online to complete the Using a for Loop with a List Challenge

Notice that the variable `cheese` contains the value of that element in the list.

Using a `for` loop like this works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions `range` and `len`:

```
for i in range(len(cheeses)):
    cheeses[i] = cheese[i]*2
```

Note that in this version, the variable `i` is an integer and displaying `cheeses[i]` will then display the value of that element in the list.

This loop traverses the list and updates each element. `len` returns the number of elements in the list. `range` returns a list of indices from 0 to `n-1`, where n is the length of the list. Each time through the loop, `i` gets the index of the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

Go online to complete the
Using a for Loop with a List, Part 2 Challenge

A `for` loop over an empty list never executes the body:

```
for x in empty:
    print('This never happens.')
```

List Operations

The + operator concatenates lists:

Code	Return
a = [1, 2, 3] **b = [4, 5, 6]** **c = a+b** **print(c)**	`[1, 2, 3, 4, 5, 6]`

Go online to complete the
Concatenate Lists Challenge

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

Code	Result
[0]*4	`[0, 0, 0, 0]`
[1, 2, 3]*3	`[1, 2, 3, 1, 2, 3, 1, 2, 3]`

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

Go online to complete the
The Repetition Operator Challenge

The `in` operator also works on lists.

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
```

Code	Return
`'Edam' in cheeses`	`True`
`'Brie' in cheeses`	`False`

Go online to complete the
Lousy Smarch Weather Challenge

List Slices

The slice operator also works on lists:

Code	Result
`t = ['a', 'b', 'c', 'd', 'e', 'f']` `t[1:3]`	`['b', 'c']`
`t[:4]`	`['a', 'b', 'c', 'd']`
`t[3:]`	`['d', 'e', 'f']`
`t[:]`	`['a', 'b', 'c', 'd', 'e', 'f']`

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

Go online to complete the
List Slicing, Part 1 Challenge

	Go online to complete the List Slicing, Part 2 Challenge
	Go online to complete the List Slicing, Part 3 Challenge
	Go online to complete the List Slicing, Part 4 Challenge

An important note: strings are immutable, so any manipulation of strings did not affect the original string. Lists are mutable so manipulations will change the list. Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle, or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

Code	Return

`t = ['a', 'b', 'c', 'd', 'e', 'f']`
`t[1:3] = ['x', 'y']`
`print(t)`
	`['a', 'x', 'y', 'd', 'e', 'f']`

List Methods

Python provides methods that operate on lists. For example, append adds a new element to the end of a list:

Code	Return

`t = ['a', 'b', 'c']`
 	
`t.append('d')`
 	
`print(t)`
	`['a', 'b', 'c', 'd']`

Go online to complete the Appending Smarch Challenge

`extend` takes a list as an argument and appends all of the elements:

Code	Return

`t1 = ['a', 'b', 'c']`
`t2 = ['d', 'e']`
 	
`t1.extend(t2)`
 	
`print(t1)`
	`['a', 'b', 'c', 'd', 'e']`

This example leaves `t2` unmodified.

`sort` arranges the elements of the list from low to high:

Code	Output

`t = ['d', 'c', 'e', 'b', 'a']`
 	
`t.sort()`
 	
`print(t)`
	`['a', 'b', 'c', 'd', 'e']`

Most list methods are void; they modify the list and return `None`. If you accidentally write `t = t.sort()`, you will be disappointed with the result.

Go online to complete the Sorting Months Challenge

The `sort` method sorts in ascending order by default. That is, from A-Z. You can perform a sort from Z-A using the optional argument `reverse` within the `sort` method:

Code	Output

`t = ['d', 'c', 'e', 'b', 'a']`
`t.sort(reverse=True)`
`print(t)`
	`['e', 'd', 'c', 'b', 'a']`

If you need to reverse the indices in the list (make the last element the first, make the second-to-last-element the second, etc), you can use the `reverse` method:

Code	Output

`t = ['d', 'c', 'e', 'b', 'a']`
`t.reverse()`
`print(t)`
	`['a', 'b', 'e', 'c', 'd']`

Go online to complete the
Reversing Months Challenge

Deleting Elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use pop:

Code	Output

`t = ['a', 'b', 'c', 'd', 'e']`
`x = t.pop(1)`
`print(t)`
	`['a', 'c', 'd', 'e']`

`print(x)`	B

pop modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

Go online to complete the
Using the pop Method Challenge

If you know the element you want to remove (but not the index), you can use `remove`:

Code	Output

`t = ['a', 'b', 'c', 'd', 'e']`
`t.remove('b')`
`print(t)`
	`['a', 'c', 'd', 'e']`

Go online to complete the Removing Smarch Challenge

The return value from remove is None.

If you don't need the removed value, you can use the `del` operator:

Code	Output

`t = ['a', 'b', 'c', 'd', 'e']`
`del t[1]`
`print(t)`
	`['a', 'c', 'd', 'e']`

To remove more than one element, you can use `del` with a slice index:

Code	Output

`t = ['a', 'b', 'c', 'd', 'e']`
`del t[1:4]`
`print(t)`
	`['a', 'e']`

As usual, the slice selects all the elements up to, but not including, the second index.

Go online to complete the Using the del Function Challenge

Lists and Strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use list:

Code	Output

```
s = 'spam'                              ['s', 'p', 'a', 'm']
t = list[s]
print(t)
```

Because `list` is the name of a built-in function, you should avoid using it as a variable name. I also avoid the letter l because it looks too much like the number 1. So that's why I use t.

The `list` function breaks a string into individual letters. If you want to break a string into words, you can use the `split` method:

Code	Output

`s = 'pining for the fjords'`
`t = s.split()`
`print(t)`
	`['pining', 'for', 'the', 'fjords']`

Once you have used `split` to break the string into a list of words, you can use the index operator (square bracket) to look at a particular word in the list.

Code	Output
`print(t[2])`	`the`

You can call `split` with an optional argument called a delimiter that specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

Code	Output

`s = 'spam-spam-spam'`
`t = s.split('-')`
`print(t)`
	`['spam', 'spam', 'spam']`

`join` is the inverse of `split`. It takes a list of strings and concatenates the elements. `join` is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

Code	Result

`t = ['pining', 'for', 'the', 'fjords']`
`delimiter = ' '`
`delimiter.join(t)`
	`pining for the fjords`

In this case the delimiter is a space character, so `join` puts a space between words. To concatenate strings without spaces, you can use the empty string, '', as a delimiter.

Aliasing

If a refers to an object and you assign b = a, then both variables refer to the same object:

Code	Result

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

The association of a variable with an object is called a ***reference***. In this example, there are two references to the same object: both a and b point to the same list.

An object with more than one reference has more than one name, so we say that the object is ***aliased***.

If the aliased object is mutable, changes made with one alias affect the other:

Code	Output

`a = [1, 2, 3]`
 	
`b = a`
 	
`b[0] = 17`
 	
`print(a)`
`print(b)`
	`[17, 2, 3]`
	`[17, 2, 3]`

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

Code	Output

`a = 'banana'`
 	
`b = a`
 	
`b = 'apple'`
 	
`print(a)`
`print(b)`
	`banana`
	`apple`

It almost never makes a difference whether a and b refer to the same string or not.

List Arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, `delete_head` removes the first element from a list:

```
def delete_head(t):
    del t[0]
```

Here's how it is used:

Code	Output
letters = ['a', 'b', 'c'] **delete_head(letters)** **print(letters)**	['b', 'c']

The parameter `t` and the variable `letters` are aliases for the same object.

It is important to distinguish between operations that *modify* lists and operations that *create* new lists. For example, the append method modifies a list, but the + operator creates a new list:

Code	Output
t1 = [1, 2] **t2 = t1.append(3)** **print(t1)**	[1, 2, 3]
print(t2)	None
t3 = t1 + [3] **print(t3)**	[1, 2, 3]
t2 is t3	False

This difference is important when you write functions that are supposed to modify lists. For example, this function does not delete the head of a list:

```
def bad_delete_head(t):
    t = t[1:]              # WRONG!
```

The `slice` operator creates a new list and the assignment makes `t` refer to it, but none of that has any effect on the list that was passed as an argument.

An alternative is to write a function that creates and returns a new list. For example, `tail` returns all but the first element of a list:

```
def tail(t):
    return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

Code	Output

`letters = ['a', 'b', 'c']`
`rest = tail(letters)`
`print(rest)`
	`['b', 'c']`

Debugging Lists

Careless use of lists (and other mutable objects) can lead to long hours of debugging. Here are some common pitfalls and ways to avoid them:

1. Don't forget that most list methods modify the argument and return None. This is the opposite of the string methods, which return a new string and leave the original alone.

If you are used to writing string code like this:

```
word = word.strip()
```

It is tempting to write list code like this:

```
t = t.sort()            # WRONG!
```

Because sort returns None, the next operation you perform with `t` is likely to fail.

Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode.

2. Pick an idiom and stick with it.

Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use pop, `remove`, `del`, or even a `slice` assignment.

To add an element, you can use the append method or the + operator. But don't forget that these are right:

```
t.append(x)
t = t + [x]
```

And these are wrong:

```
t.append([x])           # WRONG!
t = t.append(x)         # WRONG!
t + [x]                 # WRONG!
t = t + x               # WRONG!
```

Try out each of these examples in interactive mode to make sure you understand what they do. Notice that only the last one causes a runtime error; the other three are legal, but they do the wrong thing.

3. Make copies to avoid aliasing.

If you want to use a method like sort that modifies the argument, but you need to keep the original list as well, you can make a copy.

```
orig = t[:]
t.sort()
```

In this example you could also use the built-in function sorted, which returns a new, sorted list and leaves the original alone. But in that case you should avoid using sorted as a variable name!

Dictionaries

A ***dictionary*** is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type.

You can think of a dictionary as a mapping between a set of indices (which are called ***keys***) and a set of ***values***. Each key maps to a value. The association of a key and a value is called a key-value pair or sometimes an item.

As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

The function `dict` creates a new dictionary with no items. Because `dict` is the name of a built-in function, you should avoid using it as a variable name.

Code	Output
`engl_2_span = dict()` `print(engl_2_span)`	`{}`

The curly brackets, `{}`, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```
engl_2_span['one'] = 'uno'
```

This line creates an item that maps from the key 'one' to the value 'uno'. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

Code	Output
`print(engl_2_span)`	`{'one': 'uno'}`

This output format is also an input format. For example, you can create a new dictionary with three items:

```
engl_2_span = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

But if you print `engl_2_span`, you might be surprised:

Code	Output
`print(engl_2_span)`	`{'one': 'uno', 'three': 'tres', 'two': 'dos'}`

The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

Code	Output
`print(engl_2_span['two'])`	`dos`

The key `'two'` always maps to the value `'dos'` so the order of the items doesn't matter.

Go online to complete the Create a Dictionary Challenge

If the key isn't in the dictionary, you get an exception:

Code	Output
`print(engl_2_span['four'])`	`KeyError: 'four'`

The `in` operator works on dictionaries; it tells you whether something appears as a *key* in the dictionary (appearing as a value is not good enough).

Code	Result
`'one' in engl_2_span`	`True`
`'uno' in engl_2_span`	`False`

Go online to complete the Display and Check Dictionary Elements Challenge

To see whether something appears as a value in a dictionary, you can use the method `values`, which returns the values as a list, and then use the `in` operator:

Code	Result
`vals = engl_2_span.values()` `'uno' in vals`	`True`

The `in` operator uses different algorithms for lists and dictionaries.

- For lists, it uses a *linear search algorithm*. As the list gets longer, the search time gets longer in direct proportion to the length of the list.
- For dictionaries, Python uses an algorithm called a *hash table* that has a remarkable property—the `in` operator takes about the same amount of time no matter how many items there are in a dictionary. I won't explain why hash functions are so magical, but you can read more about it at http://wikipedia.org/wiki/Hash_table.

Dictionaries and Functions

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

Code	Result
`len(engl_2_span)`	`3`

The `min` and `max` functions also work on dictionaries, but they look at the minimum and maximum value of the key and not the value. Given:

```
d = {'a':10, 'b':5}
```

Code	Result
`min(d)`	`'a'`
`max(d)`	`'b'`

So notice that even though the value of `d['a']` is a larger number than the value of `d['b']`, the `min` and `max` functions returned the greatest and lowest values of the keys instead.

Looping and Dictionaries

If you use a dictionary as the sequence in a `for` statement, it traverses the keys of the dictionary. This loop prints each key and the corresponding value:

```
counts = {'parrot':1 , 'cheese':42, 'spam':100}

for key in counts:
    print(key, counts[key])
```

Here's what the output looks like:

```
spam 100
parrot 1
cheese 42
```

Again, the keys are in no particular order.

We can use this pattern to implement the various loop idioms that we have described earlier. For example if we wanted to find all the entries in a dictionary with a value above ten, we could write the following code:

```
counts = { 'parrot' : 1 , 'cheese' : 42, 'spam': 100}

for key in counts:
    if counts[key] > 10 :
        print(key, counts[key])
```

The `for` loop iterates through the keys of the dictionary, so we must use the index operator to retrieve the corresponding value for each key. Here's what the output looks like:

```
spam 100
cheese 42
```

We see only the entries with a value above 10.

Go online to complete the Water, anyone? Challenge

If you want to print the keys in alphabetical order, you first make a list of the keys in the dictionary using the keys method available in dictionary objects, and then sort that list and loop through the sorted list, looking up each key and printing out key-value pairs in sorted order as follows:

```
counts = {'parrot' : 1 , 'cheese' : 42, 'spam': 100}

key_list = counts.keys()

print(key_list)

key_list.sort()

for key in key_list:
    print(key, counts[key])
```

Here's what the output looks like:

```
['spam', 'parrot', 'cheese']
cheese 42
parrot 1
spam 100
```

First you see the list of keys in unsorted order that we get from the keys method. Then we see the key-value pairs in order from the `for` loop.

Dictionary as a Set of Counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function ord), use the number as an index into the list, and increment the appropriate counter.
3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An implementation is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```
word = 'brontosaurus'

dictionary = dict()

for letter in word:
    if letter not in dictionary:
        dictionary[letter] = 1

    else:
        dictionary[letter] = dictionary[letter] + 1

print(dictionary)
```

We are effectively computing a *histogram*, which is a statistical term for a set of counters (or frequencies).

The `for` loop traverses the string. Each time through the loop, if the character `letter` is not in the dictionary, we create a new item with key `letter` and the initial value 1 (since we have seen this letter once). If `letter` is already in the dictionary we increment `dictionary[letter]`.

Here's the output of the program:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters 'a' and 'b' appear once; 'o' appears twice, and so on.

Dictionaries have a method called `get` that takes a key and a default value. If the key appears in the dictionary, `get` returns the corresponding value; otherwise it returns the default value. For example:

Code	Result
`counts = {'parrot':1, 'cheese':42, 'spam':100}` **`print(counts.get('spam', 0))`**	`100`
`print(counts.get('tim', 0))`	`0`

We can use get to write our histogram loop more concisely. Because the `get` method automatically handles the case where a key is not in a dictionary, we can reduce four lines down to one and eliminate the `if` statement.

```
word = 'brontosaurus'

dictionary = dict()

for letter in word:
    dictionary[letter] = dictionary.get(letter,0) + 1

print(dictionary)
```

The use of the `get` method to simplify this counting loop ends up being a very commonly used "idiom" in Python and we will use it many times. So you should take a moment and compare the loop using the `if` statement and `in` operator with the loop using the `get` method. They do exactly the same thing, but one is more succinct.

Using if and in:

```
if letter not in dictionary:
    dictionary[letter] = 1
```

```
else:
    dictionary[letter] = dictionary[letter] + 1
```

Using get:

```
dictionary[letter] = dictionary.get(letter,0) + 1
```

Debugging Dictionaries

As you work with bigger datasets it can become unwieldy to debug by printing and checking data by hand. Here are some suggestions for debugging large datasets:

1. Scale down the input:

If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files themselves, or (better) modify the program so it reads only the first n lines.

If there is an error, you can reduce n to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.

2. Check summaries and types:

Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers.

A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.

3. Write self-checks:

Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a "sanity check" because it detects results that are "completely illogical".

Another kind of check compares the results of two different computations to see if they are consistent. This is called a "consistency check".

4. Pretty print the output:

Formatting debugging output can make it easier to spot an error.

Again, time you spend building scaffolding can reduce the time you spend debugging.

Tuples

Tuples are used to hold together multiple objects. Think of them as similar to lists, but without the extensive functionality that the list class gives you. One major feature of tuples is that they are immutable like strings i.e. you cannot modify tuples.

Tuples are defined by specifying items separated by commas within an optional pair of parentheses.

Tuples are usually used in cases where a statement or a user-defined function can safely assume that the collection of values i.e. the tuple of values used will not change.

Fun fact: The word "tuple" comes from the names given to sequences of numbers of varying lengths: single, double, triple, quadruple, quituple, sextuple, septuple, etc.

Tuples are Immutable

A tuple is a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are immutable. Tuples are also comparable and hashable so we can sort lists of them and use tuples as key values in Python dictionaries.

Syntactically, a tuple is a comma-separated list of values:

```
t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common and a best practice to enclose tuples in parentheses to help us quickly identify tuples when we look at Python code:

```
t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include the final comma:

Code	Result
t1 = ('a',) **type(t1)**	<type 'tuple'>

Without the comma Python treats `('a')` as an expression with a string in parentheses that evaluates to a string:

Code	Result
t2 = ('a') **type(t2)**	<type 'str'>

Go online to complete the
Create a Tuple Challenge

Another way to construct a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

Code	Output
`t = tuple()` `print(t)`	`()`

If the argument is a sequence (string, list, or tuple), the result of the call to tuple is a tuple with the elements of the sequence:

Code	Output
`t = tuple('lupins')` `print(t)`	`('l', 'u', 'p', 'i', 'n', 's')`

Because tuple is the name of a constructor, you should avoid using it as a variable name.

Most list operators also work on tuples. The bracket operator indexes an element:

Code	Output
`t = ('a', 'b', 'c', 'd', 'e')` `print(t[0])`	`a`

And the `slice` operator selects a range of elements.

Code	Output
`t = ('a', 'b', 'c', 'd', 'e')` `print(t[1:3])`	`('b', 'c')`

Go online to complete the
Slice a Tuple Challenge

But if you try to modify one of the elements of the tuple, you get an error:

Code	Output
`t = ('a', 'b', 'c', 'd', 'e')` `t[0] = 'A'`	`TypeError: 'tuple' object does not support item assignment`

You can't modify the elements of a tuple, but you can replace one tuple with another:

Code	Output

`t = ('a', 'b', 'c', 'd', 'e')`
`t = ('A',) + t[1:]`
`print(t)`
	`('A', 'b', 'c', 'd', 'e')`

Comparing Tuples

The comparison operators work with tuples and other sequences. Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

Code	Result
`(0, 1, 2) < (0, 3, 4)`	`True`
`(0, 1, 2000000) < (0, 3, 4)`	`True`

The `sort` function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on.

This feature lends itself to a pattern called DSU for:

Decorate: a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,

Sort: the list of tuples using the Python built-in sort, and

Undecorate: by extracting the sorted elements of the sequence.

For example, suppose you have a list of words and you want to sort them from longest to shortest:

```
txt = 'Tis but a flesh wound'

words = txt.split()

t = list()

for word in words:
   t.append((len(word), word))
```

```
t.sort(reverse=True)

res = list()

for length, word in t:
    res.append(word)

print(res)
```

Code	Output	Notes
`txt = 'Tis but a flesh wound'`	*None*	We create the string `txt`.
`words = txt.split()`	*None*	The `split` method uses the default argument to create a list of individual words.
`t = list()`	*None*	We create an empty list called `t`.
`for word in words:` **`    t.append((len(word), word))`**	*None*	Iterate through the list words. Append to the list `t` a tuple which contains the length of each word and the word itself.
`t.sort(reverse=True)`	*None*	`sort` compares the first element (length) first, and only considers the second element to break ties. The keyword argument `reverse=True` tells sort to go in decreasing order.
`res = list()`	*None*	We create an empty list called `res`.
`for length, word in t:` **`    res.append(word)`**	*None*	The second loop traverses the list of tuples and builds a list of words in descending order of length. The four-character words are sorted in reverse alphabetical order, so "what" appears before "soft" in the following list.
`print(res)`	`['wound', 'flesh', 'but', 'Tis', 'a']`	We display the resulting list.

Tuple Assignment

One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement. This allows you to assign more than one variable at a time when the left side is a sequence.

In this example we have a two-element list (which is a sequence) and assign the first and second elements of the sequence to the variables x and y in a single statement.

```
m = ['have', 'fun']

x, y = m
```

Code	Output
print(x)	have
print(y)	fun

It is not magic, Python roughly translates the tuple assignment syntax to be the following:

```
m = ['have', 'fun']

x = m[0]

y = m[1]
```

Code	Output
print(x)	have
print(y)	fun

It should be noted that Python does not translate the syntax literally. For example, if you try this with a dictionary, it will not work as you might expect.

Stylistically when we use a tuple on the left side of the assignment statement, we omit the parentheses, but the following is an equally valid syntax:

```
m = ['have', 'fun']

(x, y) = m
```

Code	Output
print(x)	have
print(y)	fun

A particularly clever application of tuple assignment allows us to swap the values of two variables in a single statement:

```
a, b = b, a
```

Both sides of this statement are tuples, but the left side is a tuple of variables; the right side is a tuple of expressions. Each value on the right side is assigned to its respective variable on the left side. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right must be the same:

Code	Result
`a, b = 1, 2, 3`	`ValueError: too many values to unpack (expected 2)`

More generally, the right side can be any kind of sequence (string, list, or tuple). For example, to split an email address into a user name and a domain, you could write:

```
addr = 'monty@python.org'

uname, domain = addr.split('@')
```

The return value from `split` is a list with two elements; the first element is assigned to `uname`, the second to `domain`.

Code	Output
`print(uname)`	`monty`
`print(domain)`	`python.org`

Dictionaries and Tuples

Dictionaries have a method called `items` that returns a list of tuples, where each tuple is a key-value pair.

Code	Output

`d = {'a':10, 'b':1, 'c':22}`
`t = list(d.items())`
`print(t)`
	`[('a', 10), ('c', 22), ('b', 1)]`

As you should expect from a dictionary, the items are in no particular order.

However, since the list of tuples is a list, and tuples are comparable, we can now sort the list of tuples. Converting a dictionary to a list of tuples is a way for us to output the contents of a dictionary sorted by key:

Code	Output
`d = {'a':10, 'b':1, 'c':22}` `t = list(d.items())`	`[('a', 10), ('c', 22), ('b', 1)]`

print(t)	
t.sort() **print(t)**	[('a', 10), ('b', 1), ('c', 22)]

The new list is sorted in ascending alphabetical order by the key value.

Go online to complete the
Sorting Ingredients Challenge

Multiple Assignment with Dictionaries

Combining items, tuple assignment, and `for`, you can see a nice code pattern for traversing the keys and values of a dictionary in a single loop:

```
for key, val in d.items():
    print val, key
```

This loop has two iteration variables because items returns a list of tuples and `key`, `val` is a tuple assignment that successively iterates through each of the key-value pairs in the dictionary.

For each iteration through the loop, both `key` and `value` are advanced to the next key-value pair in the dictionary (still in hash order).

The output of this loop is:

```
10 a
22 c
1 b
```

Again, it is in hash key order (i.e., no particular order).

If we combine these two techniques, we can print out the contents of a dictionary sorted by the value stored in each key-value pair.

To do this, we first make a list of tuples where each tuple is (value, key). The items method would give us a list of (key, value) tuples—but this time we want to sort by value, not key. Once we have constructed the list with the value-key tuples, it is a simple matter to sort the list in reverse order and print out the new, sorted list.

Code	Output	Notes
d = {'a':10, 'b':1, 'c':22}	*None*	We create the dictionary `d`.
l = list()	*None*	We create an empty list called `l`.
for key, val in d.items(): ** l.append((val, key))**	*None*	Iterate through the dictionary `d`. Append to the list `l` a tuple which contains the value and key of each

		dictionary entry.
`print(l)`	`[(10, 'a'), (22, 'c'), (1, 'b')]`	We display the resulting list of tuples.
`l.sort(reverse=True)`	*None*	`sort` compares the first element (value) first, and only considers the second element to break ties. The keyword argument `reverse=True` tells sort to go in decreasing order.
`print(l)`	`[(22, 'c'), (10, 'a'), (1, 'b')]`	We display the resulting, newly-sorted, list of tuples.

By carefully constructing the list of tuples to have the value as the first element of each tuple, we can sort the list of tuples and get our dictionary contents sorted by value.

Using Tuples as Keys in Dictionaries

Because tuples are hashable and lists are not, if we want to create a composite key to use in a dictionary we must use a tuple as the key.

We would encounter a composite key if we wanted to create a telephone directory that maps from last-name, first-name pairs to telephone numbers. Assuming that we have defined the variables last, first, and number, we could write a dictionary assignment statement as follows:

```
directory[last, first] = number
```

The expression in brackets is a tuple. We could use tuple assignment in a `for` loop to traverse this dictionary.

```
for last, first in directory:
    print(first, last, directory[last,first])
```

This loop traverses the keys in directory, which are tuples. It assigns the elements of each tuple to last and first, then prints the name and corresponding telephone number.

Keeping Them All Straight:

Quotes, parentheses, brackets, curly braces: these are important. Here is a quick chart so you know which is used for which type:

Code	Notes
`a = ''`	String
`b = ()`	Tuple

`c = {}`	Dictionary
`d = []`	List

When you access the individual elements within the sequence, you always use square bracket notation with the index or key:

Code	Output
`my_string = 'hello!'` `print(my_string[0])`	`'h'`
`my_tuple = (1, 'two')` `print(my_tuple[1])`	`two`
`my_dict = {'firstname':'Fred', 'lastname':'Weasley'}` `print(my_dict['firstname'])`	`Fred`
`my_list = ['one', 2]` `print(my_list[0])`	`one`

Sequences: Strings, Lists, and Tuples – Oh my!

Several examples focused on lists of tuples, but almost all of the examples in this lesson also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.

In many contexts, the different kinds of sequences (strings, lists, and tuples) can be used interchangeably. So how and why do you choose one over the others?

To start with the obvious, **strings** are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

- In some contexts, like a return statement, it is syntactically simpler to create a tuple than a list. In other contexts, you might prefer a list.
- If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.
- If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

Because **tuples** are immutable, they don't provide methods like sort and reverse, which modify existing lists. However Python provides the built-in functions sorted and reversed, which take any sequence as a parameter and return a new sequence with the same elements in a different order.

Debugging Data Structures

Lists, dictionaries and tuples are known generically as data structures; in this chapter we are starting to see compound data structures, like lists of tuples, and dictionaries that contain tuples as keys and lists as values. Compound data structures are useful, but they are prone to what I call shape errors; that is, errors caused when a data structure has the wrong type, size, or composition, or perhaps you write some code and forget the shape of your data and introduce an error.

For example, if you are expecting a list with one integer and I give you a plain old integer (not in a list), it won't work.

When you are debugging a program, and especially if you are working on a hard bug, there are four things to try:

reading: Examine your code, read it back to yourself, and check that it says what you meant to say.

running: Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to spend some time to build scaffolding.

ruminating: Take some time to think! What kind of error is it: syntax, runtime, semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?

retreating: At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can start rebuilding.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode.

For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall into a pattern I call "random walk programming", which is the process of making random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

You have to take time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

Taking a break helps with the thinking. So does talking. If you explain the problem to someone else (or even to yourself), you will sometimes find the answer before you finish asking the question.

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that works and that you understand.

Beginning programmers are often reluctant to retreat because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can paste the pieces back in a little bit at a time.

Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.