Python 4

Lists and Tuples

Lists

Lists are data types that store a collection of data.

- Lists are used to store an ordered, *indexed* collection of data.
- Values are separated by commas
- Values are enclosed in square brackets '[]'
- Lists can grow and shrink
- Values are mutable

```
[ 'atg' , 'aaa' , 'agg' ]
```

Tuples

- Tuples are used to store an ordered, *indexed* collection of data.
- Values are separated by commas
- Values are enclosed in parentheses '()'
- Tuples can **NOT** grow or shrink
- Values are immutable

```
( 'Jan' , 'Feb' , 'Mar' , 'Apr' , 'May' , 'Jun' , 'Jul' , 'Aug' , 'Sep' , 'Oct' , 'Nov' ,
'Dec' )
```

Many functions and methods return tuples like `math.modf(x)`. This function returns the fractional and integer parts of `x` in a two-item tuple. There is no reason to change this sequence.

```
>>> math.modf(2.6)
(0.6000000000000001, 2.0)
```

Back to Lists

Accessing Values in Lists

To retrieve a single value in a list use the value's index in this format list[index]. This will return the value at the specified index, starting with 0.

Here is a list:

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
```

> There are 3 values with the indices of 0, 1, 2

Index	Value
0	atg
1	aaa
2	agg

Let's access the 0th value, this is the element in the list with index 0. You'll need an index number (`0`) inside square brackets like this `[0]` . This goes after the name of the list (`codons`)

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> codons[0]
'atg'
```

The value can be saved for later use by storing in a variable.

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> first_codon = codons[0]
>>> print(first_codon)
atg
```

> Each value can be saved in a new variable to use later.

The values can be retrieved and used directly.

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> print(codons[0])
atg
>>> print(codons[1])
aaa
>>> print(codons[2])
agg
```

> The 3 values are independently accessed and immediately printed. They are not stored in a variable.

If you want to access the values starting at the end of the list, use negative indices.

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> print(codons[-1])
agg
>>> print(codons[-2])
aaa
```

> Using a negative index will return the values from the end of the list. For example, -1 is the index of the last value 'agg'. This value also has an index of 2.

Changing Values in a List

Individual values can be changed using the value's index and the assignment operator.

```
>>> print(codons)
['atg', 'aaa', 'agg']
>>> codons[2] = 'cgc'
>>> print(codons)
['atg', 'aaa', 'cgc']
```

What about trying to assign a value to an index that does not exist?

```
>>> codons[5] = 'aac'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

> codon[5] does not exist, and when we try to assign a value to this index we get an IndexError. If you want to add new elements to the end of a list use `codons.append('taa')` or `codons.extend(list)`. See below for more details.

Extracting a Subset of a List, or Slicing

This works in exactly the same way with lists as it does with strings. This is because both are sequences, or ordered collections of data with positional information. Remember Python counts the divisions between the elements, starting with 0.

Index	Value
0	atg
1	aaa
2	agg
3	aac
4	cgc
5	acg

use the syntax [start : end : step] to slice and dice your python seqeunce

```
>>> codons = [ 'atg' , 'aaa' , 'agg' , 'aac' , 'cgc' , 'acg']
>>> print (codons[1:3])
['aaa', 'agg']
>>> print (codons[3:])
['aac', 'cgc', 'acg']
>>> print (codons[:3])
['atg', 'aaa', 'agg']
>>> print (codons[0:3])
['atg', 'aaa', 'agg']
```

> `codons[1:3]` returns every value starting with the value of codons[1] up to but not including codons[3]
> `codons[3:]` returns every value starting with the value of codons[3] and every value after.
> `codons[:3]` returns every value up to but not including codons[3]
> `codons[0:3]` is the same as `codons[:3]`

List Operators

Operator	Description	Example
`+`	Concatenation	`[10, 20, 30] + [40, 50, 60]` returns `[10, 20, 30, 40, 50, 60]`
`*`	Repetition	`['atg'] * 4` returns `['atg','atg','atg','atg']`
`in`	Membership	`20 in [10, 20, 30]` returns `True`

List Functions

Functions	Description	Example
`len(list)`	returns the length or the number of values in list	`len([1,2,3])` returns `3`
`max(list)`	returns the value with the highest ASCII value (=latest in ASCII alphabet)	`max(['a','A','z'])` returns `'z'`
`min(list)`	returns the value with the lowest ASCII value (=earliest in ASCII alphabet)	`min(['a','A','z'])` returns `'A'`
`list(seq)`	converts a tuple into a list	`list(('a','A','z'))` returns `['a', 'A', 'z']`
`sorted(list, key=None, reverse=False)`	returns a sorted list based on the key provided	`sorted(['a','A','z'])` returns `['A', 'a', 'z']`
`sorted(list, key=str.lower, reverse=False)`	`str.lower()` makes all the elements lowercase before sorting	`sorted(['a','A','z'],key=str.lower)` returns `['a', 'A', 'z']`

List Methods

Remember methods are used in the following format list.method().

For these examples use: `nums = [1,2,3]` and `codons = [ 'atg' , 'aaa' , 'agg' ]`

Method	Description	Example
`list.append(obj)`	appends an object to the end of a list	nums.append(9) ; print(nums) ; returns [1,2,3,9]
`list.count(obj)`	counts the occurrences of an object in a list	nums.count(2) returns 1
`list.index(obj)`	returns the lowest index where the given object is found	nums.index(2) returns 1
`list.pop()`	removes and returns the last value in the list. The list is now one element shorter	nums.pop() returns 3
`list.insert(index, obj)`	inserts a value at the given index. Remember to think about the divisions between the elements	nums.insert(0,100) ; print(nums) returns [100, 1, 2, 3]
`list.extend(new_list)`	appends `new_list` to the end of `list`	nums.extend([7, 8]) ; print(nums) returns [1, 2, 3, 7,8]
`list.pop(index)`	removes and returns the value of the index argument. The list is now 1 value shorter	nums.pop(0) returns 1
`list.remove(obj)`	finds the lowest index of the given object and removes it from the list. The list is now one element shorter	codons.remove('aaa') ; print(codons) returns ['atg' , 'agg']
`list.reverse()`	reverses the order of the list	nums.reverse() ; print(nums) returns [3,2,1]
`list.copy()`	Returns a shallow copy of list. Shallow vs Deep only matters in multidimensional data structures.	
`list.sort([func])`	sorts a list using the provided function. Does not return a list. The list has been changed. Advanced list sort will be covered once writing your own functions has been discussed.	codons.sort() ; print(codons) returns ['aaa', 'agg', 'atg']

Be careful how you make a copy of your list

```
>>> my_list=['a', 'one', 'two']
>>> copy_list=my_list
>>> copy_list.append('1')
>>> print(my_list)
['a', 'one', 'two', '1']
>>> print(copy_list)
['a', 'one', 'two', '1']
```

> Not what you expected?! Both lists have changed because we only copied a pointer to the original list when we wrote `copy_list=my_list`.

Let's copy the list using the `copy()` method.

```
>>> my_list=['a', 'one', 'two']
>>> copy_list=my_list.copy()
>>> copy_list.append('1')
>>> print(my_list)
['a', 'one', 'two']
```

> There we go, we get what we expect this time!

Building a List one Value at a Time

Now that you have seen the `append()` function we can go over how to build a list one value at a time.

```
>>> words = []
>>> print(words)
[]
>>> words.append('one')
>>> words.append('two')
>>> print(words)
['one', 'two']
```

> We start with a an empty list called 'words'. We use `append()` to add the value 'one' then to add the value 'two'. We end up with a list with two values. You can add a whole list to another list with `words.extend(['three','four','five'])`

Loops

All of the coding that we have gone over so far has been executed line by line. Sometimes there are blocks of code that we want to execute more than once. Loops let us do this.

There are two loop types:

1. while loop
2. for loop

While loop

The while loop will continue to execute a block of code as long as the test expression evaluates to `True`.

While Loop Syntax

```
while expression:
  # these statements get executed every time the code enters the loop
  statement1
  statement2
  more_statements
 code below here gets executed after the while loop exits
rest_of_code_goes_here
more_code
```

> The condition is the expression. The while loop block of code is the collection of indented statements following the expression.

Code:

```
!/usr/bin/env python3

count = 0
while count < 5:
  print("count:" , count)
  count+=1
print("Done")
```

Output:

```
$ python while.py
count: 0
count: 1
count: 2
count: 3
count: 4
Done
```

> The while condition was true 5 times and the while block of code was executed 5 times.
>
> - count is equal to 0 when we begin
> - 0 is less than 5 so we execute the while block
> - count is printed
> - count is incremented (count = count + 1)
> - count is now equal to 1.
> - 1 is less than 5 so we execute the while block for the 2nd time.

> - this continues until count is 5.
> - 5 is not less than 5 so we exit the while block
> - The first line following the while statement is executed, "Done" is printed

Infinite Loops

An infinite loop occurs when a while condition is always true. Here is an example of an infinite loop.

```
!/usr/bin/env python3

count = 0
while count < 5:            # this is normally a bug!!
  print("count:" , count)   # forgot to increment count in the loop!!
print("Done")
```

Output:

```
$ python infinite.py
count: 0
count: 0
count: 0
count: 0
count: 0
count: 0
count: 0
count: 0
...
...
```

> What caused the expression to always be `True`?
> The statement that increments the count is missing, so it will always be smaller than 5. To stop the code from printing forever use ctrl+c. Behavior like this is almost always due to a bug in the code.

A better way to write an infinite loop is with `True`. You'll need to include something like `if ...: break`

```
!/usr/bin/env python3
count=0
while True:
  print("count:",count)
  # you probably want to add if...: break
  # so you can get out of the infinite loop
print('Finished the loop')
```

For Loops

A for loop is a loop that executes the for block of code for every member of a sequence, for example the elements of a list or the letters in a string.

For Loop Syntax

```
for iterating_variable in sequence:
  statement(s)
```

An example of a sequence is a list. Let's use a for loop with a list of words.

Code:

```
!/usr/bin/env python3

words = ['zero','one','two','three','four']
for word in words:
  print(word)
```

> Notice how I have named my variables, the list is plural and the iterating variable is singular

Output:

```
python3 list_words.py
zero
one
two
three
four
```

This next example is using a for loop to iterate over a string. Remember a string is a sequence like a list. Each character has a position. Look back at "Extracting a Substring, or Slicing" in the Strings section to see other ways that strings can be treated like lists.

Code:

```
!/usr/bin/env python3

dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
for nt in dna:
  print(nt)
```

Output:

```
$ python3 for_string.py
G
T
A
C
C
T
T
...
...
```

> This is an easy way to access each character in a string. It is especially nice for DNA sequences.

Another example of iterating over a list of variables, this time numbers.

Code:

```
!/usr/bin/env python3

numbers = [0,1,2,3,4]
for num in numbers:
  print(num)
```

Output:

```
$ python3 list_numbers.py
0
1
2
3
4
```

Python has a function called `range()` that will return numbers that can be converted to a list.

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

The function `range()` can be used in conjunction with a for loop to iterate over a range of numbers. Range also starts at 0 and thinks about the gaps between the numbers.

Code:

```
!/usr/bin/env python3

for num in range(5):
  print(num)
```

Output:

```
$ python list_range.py
0
1
2
3
4
```

> As you can see this is the same output as using the list `numbers = [0, 1, 2, 3, 4]`
> And this has the same functionality as a while loop with a condition of `count = 0` ; `count < 5`.

This is the equivalent while loop

Code:

```
count = 0
while count < 5:
  print(count)
  count+=1
```

Output:

```
0
1
2
3
4
```

PCR Program Loop Example

pcr.py

Standard PCR program

1. Initial Denature: 94°C 3 min
2. Denature: 94°C 30 sec
3. Annealing: 57°C 30 sec
4. Extension: 72°C 1 min

5. Go to step 2, for additional 29 times
6. Final Extension: 72°C 5 min
7. 4°C for ever

```
!/usr/bin/env python3

def doAnnealing(time):
  temp = 57
  print(f"  Annealing at temp {temp}oC for {time}")

def doDenature(time):
  temp = 94
  print(f"  Denaturing at temp {temp}oC for {time}")

def doExtension(time):
  temp = 72
  print(f"  Extending at temp {temp}oC for {time}")

def doChilling(time):
  temp = 4
  print(f"  Chilling at temp {temp}oC for {time}")


cycles = 30
print(f"PCR Started.")

doDenature("3min")

for cycle in range(cycles):
  cycle+=1
  print(f"Starting Cycle {cycle}")
  doDenature("30sec")
  doAnnealing("30sec")
  doExtension("1min")

doExtension("5min")

print(f"PCR Complete.")
print(f"Starting Chilling")

while (True):
 doChilling("forever")
```

Output:

```
% python pcr.py
```

```
PCR Started.
  Denaturing at temp 94oC for 3min
Starting Cycle 1
  Denaturing at temp 94oC for 30sec
  Annealing at temp 57oC for 30sec
  Extending at temp 72oC for 1min
Starting Cycle 2
  Denaturing at temp 94oC for 30sec
  Annealing at temp 57oC for 30sec
  Extending at temp 72oC for 1min
Starting Cycle 3
  Denaturing at temp 94oC for 30sec
  Annealing at temp 57oC for 30sec
  Extending at temp 72oC for 1min
Starting Cycle 4
  Denaturing at temp 94oC for 30sec
  Annealing at temp 57oC for 30sec
  Extending at temp 72oC for 1min
Starting Cycle 5
  Denaturing at temp 94oC for 30sec
  Annealing at temp 57oC for 30sec
  Extending at temp 72oC for 1min
Starting Cycle 6
  Denaturing at temp 94oC for 30sec
  Annealing at temp 57oC for 30sec
  Extending at temp 72oC for 1min
...
Starting Cycle 26
  Denaturing at temp 94oC for 30sec
  Annealing at temp 57oC for 30sec
  Extending at temp 72oC for 1min
Starting Cycle 27
  Denaturing at temp 94oC for 30sec
  Annealing at temp 57oC for 30sec
  Extending at temp 72oC for 1min
Starting Cycle 28
  Denaturing at temp 94oC for 30sec
  Annealing at temp 57oC for 30sec
  Extending at temp 72oC for 1min
Starting Cycle 29
  Denaturing at temp 94oC for 30sec
  Annealing at temp 57oC for 30sec
  Extending at temp 72oC for 1min
Starting Cycle 30
  Denaturing at temp 94oC for 30sec
  Annealing at temp 57oC for 30sec
  Extending at temp 72oC for 1min
  Extending at temp 72oC for 5min
```

```
PCR Complete.
Starting Chilling
  Chilling at temp 4oC for forever
  Chilling at temp 4oC for forever
  ...
```

Loop Control

Loop control statements allow for altering the normal flow of execution.

Control Statement	Description
`break`	A loop is terminated when a break statement is executed. All the lines of code after the break, but within the loop block are not executed. No more iteration of the loop are preformed
`continue`	A single iteration of a loop is terminated when a continue statement is executed. The next iteration will proceed normally.

Loop Control: Break

Code:

```
!/usr/bin/env python3

count = 0
while count < 5:
  print("count:" , count)
  count+=1
  if count == 3:
    break
print("Done")
```

Output:

```
$ python break.py
count: 0
count: 1
count: 2
Done
```

> when the count is equal to 3, the execution of the while loop is terminated, even though the initial condition (count < 5) is still True.

Loop Control: Continue

Code:

```
!/usr/bin/env python3

count = 0
while count < 5:
  print("count:" , count)
  count+=1
  if count == 3:
    continue
  print("line after our continue")
print("Done")
```

Output:

```
$ python continue.py
count: 0
line after our continue
count: 1
line after our continue
count: 2
count: 3
line after our continue
count: 4
line after our continue
Done
```

> When the count is equal to 3 the continue is executed. This causes all the lines within the loop block to be skipped. "line after our continue" is not printed when count is equal to 3. The next loop is executed normally.

Iterators

An iterable is any data type that is can be iterated over, or can be used in iteration. An iterable can be made into an iterator with the `iter()` function. This means you can use the `next()` function.

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> codons_iterator=iter(codons)
>>> next(codons_iterator)
'atg'
>>> next(codons_iterator)
'aaa'
>>> next(codons_iterator)
'agg'
>>> next(codons_iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

> An iterator allows you to get the next element in the iterator until there are no more elements. If you want to go through each element again, you will need to redefine the iterator.

Example of using an iterator in a for loop:

```
codons = [ 'atg' , 'aaa' , 'agg' ]
>>> codons_it = iter(codons)
>>> for codon in codons_it :
...   print( codon )
...
atg
aaa
agg
```

> This is nice if you have a large large large list that you don't want to keep in memory. An iterator allows you to go through each element but not keep the entire list in memory. Without iterators the entire list is in memory.

List Comprehension

List comprehension is a way to make a list without typing out each element. There are many many ways to use list comprehension to generate lists. Some are quite complex, yet useful.

Here is an simple example:

```
>>> dna_list = ['TAGC', 'ACGTATGC', 'ATG', 'ACGGCTAG']
>>> lengths = [len(dna) for dna in dna_list]
>>> lengths
[4, 8, 3, 8]
```

This is how you could do the same with a for loop:

```
>>> lengths = []
>>> dna_list = ['TAGC', 'ACGTATGC', 'ATG', 'ACGGCTAG']
>>> for dna in dna_list:
...   lengths.append(len(dna))
...
>>> lengths
[4, 8, 3, 8]
```

Using conditions:

This will only return the length of an element that starts with 'A':

```
>>> dna_list = ['TAGC', 'ACGTATGC', 'ATG', 'ACGGCTAG']
>>> lengths = [len(dna) for dna in dna_list if dna.startswith('A')]
>>> lengths
[8, 3, 8]
```

> This generates the following list: [8, 3, 8]

Here is an example of using mathematical operators to generate a list:

```
>>> two_power_list = [2 ** x for x in range(10)]
>>> two_power_list
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

> This creates a list of the of the product of [2^0 , 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8, 2^9]
