Tuples

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, and hence can be used as keys for a dictionary.

A tuple can be created as a comma-separated list of values:

In [1]:
```
serine_codons = 'tca', 'tcc', 'tcg', 'tct'
```

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

In [2]:
```
serine_codons = ('tca', 'tcc', 'tcg', 'tct')
```

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

In [3]:
```
methionine_codon = 'aug',
```

What type of object is created if you omit the final comma?

You can also create a tuple using the **tuple** keyword:

In [13]:
```
serine_codons = tuple(('tca', 'tcc', 'tcg', 'tct'))
serine_codons
```

Out[13]:
```
('tca', 'tcc', 'tcg', 'tct')
```

Similarly, you can create dictionaries using the **dict** keyword:

In [11]:
```
d = dict({1:'two', 2:'two'})
d
```

Out[11]:
```
{1: 'two', 2: 'two'}
```

And you can use the **list** keyword to creat a list:

In [12]:
```
l = list([1,2])
l
```

Out[12]:
```
[1, 2]
```

You can convert a tuple to a list using the `list()` function:

In [14]:
```
l = list((1,2))
l
```

Out[14]:
```
[1, 2]
```

And vice versa:

In [15]:
```
t = tuple([1,2])
t
```

Out[15]:
```
(1, 2)
```

Notice the delimiter around each type of object helps to define what type it is.

Tuples contain a set of elements just like lists and can be indexed exactly like lists:

In [17]:
```
serine_codons = 'tca', 'tcc', 'tcg', 'tct'
serine_codons[1:3]
```

Out[17]:
```
('tcc', 'tcg')
```

But the key difference betweeen lists and tuples is that tuples can not be modified but lists can:

In [20]:
```
t = ('atg', 'yga')
l = ['atg', 'yga']

l[1] = 'tga'
l
```

Out[20]:
```
['atg', 'tga']
```

Tuple assignment

In Python you can have a tuple on the left hand side of an assignment statement. For example:

In [25]:
```
seq1,seq2 = ['ATG', 'TGA']
seq1,seq2
nt1,nt2 = 'AT'
nt2
```

Out[25]:
```
'T'
```

The objects on the right can be a tuple, a list, or even a dictionary (in which case keys will be assigned to the tuple, keep in mind that dictionaries prior to python 3.6 were not ordered).

A nice application of tuple assignment is for swapping the values of two variables:

In [28]:
```
seq1,seq2 = ['ATG', 'TGA']
seq1, seq2 = seq2, seq1
seq1, seq2
```

Out[28]:
```
('TGA', 'ATG')
```

Tuples are also very useful when you have a function that needs to return multiple values:

In [29]:
```
def returns_two_values():
    return 'first', 'second'
print(type(returns_two_values()))
```

```
<class 'tuple'>
```

Here's another example of unpacking the results of a `split` operation:

In [31]:
```
sequence = 'ATG\tTGA'
seq1, seq2 = sequence.split('\t')
seq1, seq2 # seq1, seq2 is a tuple
seq1 # seq1 and seq2 are strings
```

Out[31]:
```
'ATG'
```

Exercise 10a

Assign each of the values of the following line of a csv file to a tuple.

In [35]:
```
line = '1.23,2.46,4.12'
t1, t2, t3 = line.split(',')
#t = line.split(',')
t = t1, t2, t3
t
```

Out[35]:
```
('1.23', '2.46', '4.12')
```

Comparing and sorting tuples

The various 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 ignored.

For example:

In [37]:
```
(1, 2) < (2, 1)
```

Out[37]:
```
False
```

Let's use tuples to sort a list of sequences by their length:

In [40]:
```
sequences = ['aaag', 'tctatt', 'acgttttt', 'act']
#decorate each seqeunce with its length
s = []
for sequence in sequences:
    s.append((len(sequence), sequence))
print('decorated list:', s)
s.sort()
print('sorted decorated list:', s)
sorted_sequences = []
for length, sequence in s:
    sorted_sequences.append(sequence)
sorted_sequences
```

```
decorated list: [(4, 'aaag'), (6, 'tctatt'), (8, 'acgttttt'), (3, 'act
')]
sorted decorated list: [(3, 'act'), (4, 'aaag'), (6, 'tctatt'), (8, 'a
cgttttt')]
```

Out[40]:
```
['act', 'aaag', 'tctatt', 'acgttttt']
```

Tuple Methods

Tuples have only two methods: `tuple.count(element)` and `tuple.index(element)`. But some built-in functions, such as `len()` can also be used on tuples.

In [43]:
```
t = ('G','A', 'C', 'G', 'G')
t.index('A')
len(t)
```

Out[43]:
```
5
```

Tuples and dictionaries

Because tuples are immutable they can serve as keys in a dictionary:

In [46]:
```
seqs = {}
seqs[('ATG', 'AUG')] = 'Start'
('ATG', 'AUG') in seqs
```

Out[46]:
```
True
```

In [47]:
```
fasta = {}
fasta[('>seq1', 3)] = 'ATGTAGCTGACTAC'
fasta[('>seq2', 6)] = 'ATACGTCAGGGGGG'
for key1, key2 in fasta:
    print(key1 + ' reads:' + str(key2) + '\n' + fasta[(key1, key2)])
```

```
>seq1 reads:3
ATGTAGCTGACTAC
>seq2 reads:6
ATACGTCAGGGGGG
```

Dictionaries can be converted to lists of tuples using the `dictionary.items()` method:

In [49]:
```
d = {'ATG': 3, 'TGA': 5, 'CCC': 1}
d.items()
l = list(d.items())
l
```

Out[49]:
```
[('ATG', 3), ('TGA', 5), ('CCC', 1)]
```

We can combine this with a for loop to get the keys and values for a dictionary:

In [50]:
```
d = {'ATG': 3, 'TGA': 5, 'CCC': 1}
for key, value in d.items():
    print(key, value)
```

```
ATG 3
TGA 5
CCC 1
```

Tuple or list?

In some cases tuples and lists can be used interchangeably. So how to choose which one to use?

Lists have more functionality than tuples mostly because they are mutable. But there are a few cases where tuples are a better choice:

- In a `return` statement.
- As dictionary keys.
- 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 have methods like sort and reverse, which modify existing lists. However, the built-in functions `sorted` and `reversed`, which take any sequence as a parameter and return a new sequence with the same elements in the desired order, can be used on tuples.

Let's take a look at reversed function in use on a string:

In [52]:
```
seq = 'ATG'
for nt in reversed('ATG'):
    print(nt, end='')
seq
```

```
GTA
```

Out[52]:
```
'ATG'
```

We can also use it on a tuple:

In [56]:
```
seq = ('ATG', 'TGA')
rev_seqs = ()
for t in reversed(seq):
    rev_seqs += (t,)
rev_seqs
```

Out[56]:
```
('TGA', 'ATG')
```

Recall that we can use slicing on tuples. Thus, an alternative wasy to reverse a tuple is with slicing:

In [58]:
```
seq = ('ATG', 'TGA')
rev_seqs = seq[::-1]
rev_seqs
```

Out[58]:
```
('TGA', 'ATG')
```

Exercise 10b

Use the `sorted()` function to sort the following tuple numerically (you will have to create a new tuple). The syntax is `sorted(iterable, reverse=False)`. Note that `sorted()` returns a list.

In [61]:
```
t = (1, 4, 7, 2, 3)
t_new = tuple(sorted(t))
t_new
```

Out[61]:
```
(1, 2, 3, 4, 7)
```

What happens if you use `sorted()` on a dictionary?

In [62]:
```
d = {2: 'two', 1: '1'}
sorted(d)
```

Out[62]:
```
[1, 2]
```

What happens if you use `sorted()` on a list?

In [63]:
```
l = [1, 2, 4, 3]
sorted(l)
```

Out[63]:
```
[1, 2, 3, 4]
```

What if you use the `sort()` method on a list?

In [64]:
```
l.sort()
l
```

Out[64]:
```
[1, 2, 3, 4]
```

Function arguments revisited

Consider the following function definition:

In [65]:
```
def seq_info(sequence, type_seq, source):
    print("sequence: " + sequence + ", type: " + type_seq + ", source: "
seq_info("ACTG", 'DNA', 'unknown')
```

```
sequence: ACTG, type: DNA, source: unknown
```

Recall that we can assign keword arguments:

In [66]:
```
def seq_info(sequence, type_seq='DNA', source='unknown'):
    print("sequence: " + sequence + ", type: " + type_seq + ", source: "

seq_info("ACTG")
```

```
sequence: ACTG, type: DNA, source: unknown
```

These keywords allow us to reference specific arguments when we call the function:

In [67]:
```
def seq_info(sequence, type_seq='DNA', source='unknown'):
    print("sequence: " + sequence + ", type: " + type_seq + ", source: " +

seq_info("ACTG")
seq_info("ACUG", type_seq='RNA')
seq_info("ACTG", source='Arabidopsis')
```

```
sequence: ACTG, type: DNA, source: unknown
sequence: ACUG, type: RNA, source: unknown
sequence: ACTG, type: DNA, source: Arabidopsis
```

You can learn more about **keyword arguments** in the Python tutorial (https://docs.python.org/3.6/tutorial/controlflow.html#keyword-arguments).

List Comprehensions

List comprehensions provide a simple way to create lists.

A list comprenesion assigns a new list to an expression on an old list contained within brackets. For example:

```
new_list = [expression for item in old_list if conditional]
```

The conditional statement is often optional.

Let's multiply all the elements in the list below by 2 and append them to a new list:

In [68]:
```
old_list = [1,2,3,4]
new_list = []
for n in old_list:
    new_list.append(n*2)
new_list
```

Out[68]:
```
[2, 4, 6, 8]
```

A more concise way to do this is with a list comprehension:

In [69]:
```
old_list = [1,2,3,4]
new_list = [n*2 for n in old_list]
new_list
```

Out[69]:
```
[2, 4, 6, 8]
```

Let's add a conditional statement to exclude odd numbers:

In [74]:
```
old_list = [1,2,3,4]
new_list = [n*2 for n in old_list if n % 2 == 0]
new_list
```

Out[74]:
```
[4, 8]
```

Exercise 10c

Write a function, `read_thresh(seq_list, threshold)` , that returns a new matrix that includes only elemements for which the number of reads passes some read threshold. Your function should use a list comprehension such that the only expression is within the return statement.

In [ ]:
```

```

Let's revisit the example of creating a matrix of 0s:

In [ ]:
```
def zeros_matrix(rows, columns):
    """create a matrix with a given number of rows and columns
    """
    matrix = []
    for row in range(rows) :
        matrix.append([0] * columns)
    return matrix

zeros_matrix(3,2)
```

We can do this more precisely with list comprehensions:

In [ ]:
```

```

Just for fun: lambda functions

The `lambda` function is a small anonymnous function often which is often used as an argument within a high-order function. The syntax is `lambda arguments: expression` . They can have many arguments but only one expression. They don't require a function definition and they contain an implicit return statement.

Let's write a lambda function to convert a DNA sequence to RNA:

In [ ]:
```

```

Let's write a lambda function that reverse complements a sequence:

In [ ]:

Including a bunch of lambda function will likeley makey your code confusing so they are to be used sparingly or not at all as most lambda functions can be accomplished with regular functions and list comprehensions. You will occasionally run into them in other people's code it's good to know what they are.

Additional Exercises

Answer key (http://rna.colostate.edu/dokuwiki/doku.php?id=script:ex10)

10d) Write a function, `nt_content(seq)` , that returns the numbers of As, Cs, Gs, and Ts in a DNA sequence as a tuple. Outside the function, your code should print the results, for example:

```
A: 5
C: 7
G: 6
T: 1
```

In [ ]:

10e) Using list comprehensions, reverse complement a list of sequences such as the one below.

In [ ]:
```
'ATATGCTACCCCC', 'GGGCTAGCTGAGCAAA', 'CCATGCATGCATCGGG', 'TTTTTTTTACTTCA'
```

In [ ]:

10f) Using list comprehensions, create a list of lists out of a list of tuples, such as the one below.

In [ ]:
```
tuples_list = [(1,2), (3,4), (5,6), (7,8)]
```

In [ ]: