Python 5

Dictionaries

Dictionaries are another iterable, like a string and list. They act like lists, with one big difference, instead of retrieving values with a numerical index they use strings. You can look items up in a python dictionary just like you do when you look up items in the English dictionary, the key, or a string.

For example, when you want to know what the definition of the word, 'onomatopoeia', you look it up using the word, 'onomatopoeia'. If you want to know the value, or in our example below, the sequence of TP53 you would use 'TP53' as a look up.

Dictionaries are a collection of key/value pairs. In Python, each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: `{}`

Each key in a dictionary is unique, while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Data that is appropriate for dictionaries are two pieces of information that naturally go together, like gene name and sequence.

Key	Value
TP53	GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
BRCA1	GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA

Creating a Dictionary

```
genes = { 'TP53' : 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC' ,
'BRCA1' : 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA' }
```

Breaking up the key/value pairs over multiple lines make them easier to read.

```
genes = {
            'TP53' : 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'
,
            'BRCA1' : 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
          }
```

Accessing Values in Dictionaries

To retrieve a single value in a dictionary use the value's key in this format `dict[key]`. This will return the value at the specified key.

```
>>> genes = { 'TP53' :
'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC' , 'BRCA1' :
'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA' }
>>>
>>> genes['TP53']
GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
```

> The sequence of the gene TP53 is stored as a value of the key 'TP53'. We can access the sequence by using the key in this format dict[key]

The value can be accessed and passed directly to a function or stored in a variable.

```
>>> print(genes['TP53'])
GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
>>>
>>> seq = genes['TP53']
>>> print(seq)
GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
```

Changing Values in a Dictionary

Individual values can be changed by using the key and the assignment operator.

```
>>> genes = { 'TP53' :
'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC' , 'BRCA1' :
'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA' }
>>>
>>> print(genes)
{'BRCA1': 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA', 'TP53':
'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'}
>>>
>>> genes['TP53'] = 'atg'
>>>
>>> print(genes)
{'TP53': 'atg', 'BRCA1':
'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'}
```

> The contents of the dictionary have changed.

Other assignment operators can also be used to change a value of a dictionary key.

```
>>> genes = { 'TP53' :
'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC' , 'BRCA1' :
'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA' }
>>>
>>> genes['TP53'] += 'TAGAGCCACCGTCCAGGGAGCAGGTAGCTGCTGGGCTCCGGGGACACTTTGCGTTCGGGCTGGGAGCGTG'
>>>
>>> print(genes)
{'TP53':
'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTCTAGAGCCACCGTCCAGGGAGCA
GGTAGCTGCTGGGCTCCGGGGACACTTTGCGTTCGGGCTGGGAGCGTG', 'BRCA1':
'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'}
```

> Here we have used the '+=' concatenation assignment operator. This is equivalent to `genes['TP53'] = genes['TP53'] + 'TAGAGCCACCGTCCAGGGAGCAGGTAGCTGCTGGGCTCCGGGGACACTTTGCGTTCGGGCTGGGAGCGTG'`.

Accessing Each Dictionary Key/Value

Since a dictionary is an iterable object, we can iterate through its contents.

A for loop can be used to retrieve each key of a dictionary one a time:

```
>>> for gene in genes:
...     print(gene)
...
TP53
BRCA1
```

Once you have the key you can retrieve the value:

```
>>> for gene in genes:
...     seq = genes[gene]
...     print(gene, seq[0:10])
...
TP53 GATGGGATTG
BRCA1 GTACCTTGAT
```

Building a Dictionary one Key/Value at a Time

Building a dictionary one key/value at a time is akin to what we just saw when we change a key's value. Normally you won't do this. We'll talk about ways to build a dictionary from a file in a later lecture.

```
>>> genes = {}
>>> print(genes)
{}
>>> genes['Brca1'] = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
>>> genes['TP53'] = 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'
>>> print(genes)
{'Brca1': 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA', 'TP53':
'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'}
```

> We start by creating an empty dictionary. Then we add each key/value pair using the same syntax as when we change a value.
> dict[key] = new_value

Checking That Dictionary Keys Exist

Python generates an error (NameError) if you try to access a key that does not exist.

```
>>> print(genes['HDAC'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'HDAC' is not defined
```

Dictionary Operators

Operator	Description
`in`	`key in dict` returns True if the key exists in the dictionary
`not in`	`key not in dict` returns True if the key does not exist in the dictionary

Because Python generates a NameError if you try to use a key that doesn't exist in the dictionary, you need to check whether a key exists before trying to use it.

The best way to check whether a key exists is to use `in`

```
>>> gene = 'TP53'
>>> if gene in genes:
...     print('found')
...
found
>>>
>>> if gene in genes:
...     print(genes[gene])
...
GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
>>>
```

Building a Dictionary one Key/Value at a Time using a loop

Now we have all the tools to build a dictionary one key/value using a for loop. This is how you will be building dictionaries more often in real life.

Here we are going to count and store nucleotide counts:

```
#!/usr/bin/env python3

# create a new empty dictionary
nt_count={}

# loop example from loops lecture
dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
for nt in dna:

  # is this nt in our dictionary?
  if nt in nt_count:
    # if it is, lets increment our count
    previous_count = nt_count[nt]
    new_count = previous_count + 1
    nt_count[nt] = new_count
  else:
    # if not, lets add this nt to our dictionary and make count = 1
    nt_count[nt] = 1;

print(nt_count)
```

> {'G': 20, 'T': 21, 'A': 13, 'C': 16}

What is another way we could increment our count?

```
nt_count={}

dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
for nt in dna:
  if nt in nt_count:
    nt_count[nt] += 1
  else:
    nt_count[nt] = 1;

print(nt_count)
```

> remember that `count=count+1` is the same as `count+=1`

Sorting Dictionary Keys

If you want to print the contents of a dictionary, you should sort the keys then iterate over the keys with a for loop. Why do you want to sort the keys?

```
for gene_key in sorted(genes): # python allows you to use this shortcut in a for loop
                               # you don't have to write genes.keys() in a for loop
                               # to iterate over the keys
  print(gene_key, '=>' , genes[gene_key])
```

> This will print keys in the same order every time you run your script. Dictionaries are unordered, so without sorting, you'll get a different order every time you run the script, which could be confusing.

Dictionary Functions

Function	Description
`len(dict)`	returns the total number of key/value pairs
`str(dict)`	returns a string representation of the dictionary
`type(variable)`	Returns the type or class of the variable passed to the function. If the variable is dictionary, then it would return a dictionary type.

These functions work on several other data types too!

Dictionary Methods

Method	Description
`dict.clear()`	Removes all elements of dictionary dict
`dict.copy()`	Returns a shallow copy of dictionary dict. Shallow vs. deep copying only matters in multidimensional data structures.
`dict.fromkeys(seq,value)`	Create a new dictionary with keys from seq (Python sequence type) and values set to value.
`dict.items()`	Returns a list of (key, value) tuple pairs
`dict.pop(key)`	Removes the key:value pair and returns the value
`dict.keys()`	Returns list of keys
`dict.get(key, default = None)`	get value from dict[key], use default if not present
`dict.setdefault(key, default = None)`	Similar to get(), but will set dict[key] = default if key is not already in dict
`dict.update(dict2)`	Adds dictionary dict2's key-values pairs to dict
`dict.values()`	Returns list of dictionary dict's values

Sorting a Dictionary by values in a for loop

To sort a dictionary by the values in a `for` loop use the `dict.get()` method without any arguments nested within the `sorted` function.
Remember that the `sorted()` function returns a sorted list of the given iterable object.
And that the optional key argument takes a function to provide items for the sort.

Let's sort by the alphabetical order of the sequences of genes in a dictionary:
Code:

```
genes={'Brca1': 'TTTAA', 'TP53': 'AAATT'}
for gene in sorted(genes, key=genes.get):
  print(gene, genes[gene])
```

Output:

```
TP53 AAATT
Brca1 TTTAA
```

We can reverse the sort by using `reverse=True`:
Code:

```
genes={'Brca1': 'TTTAA', 'TP53': 'AAATT'}
for gene in sorted(genes, key=genes.get, reverse=True):
  print(gene, genes[gene])
```

Output:

```
Brca1 TTTAA
TP53 AAATT
```

Sort by the length we will see later in the Functions lecture

Sets

A set is another Python data type. It is essentially a dictionary with keys but no values.

- A set is unordered
- A set is a collection of data with no duplicate elements.
- Common uses include looking for differences and eliminating duplicates in data sets.

Curly braces `{}` or the `set()` function can be used to create sets.

> Note: to create an empty set you have to use `set()`, not `{}` the latter creates an empty dictionary.

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)
{'orange', 'banana', 'pear', 'apple'}
```

> Look, duplicates have been removed

Test to see if an value is in the set

```
>>> 'orange' in basket
True
>>> 'crabgrass' in basket
False
```

> The in operator works the same with sets as it does with lists and dictionaries

Union, intersection, difference and symmetric difference can be done with sets

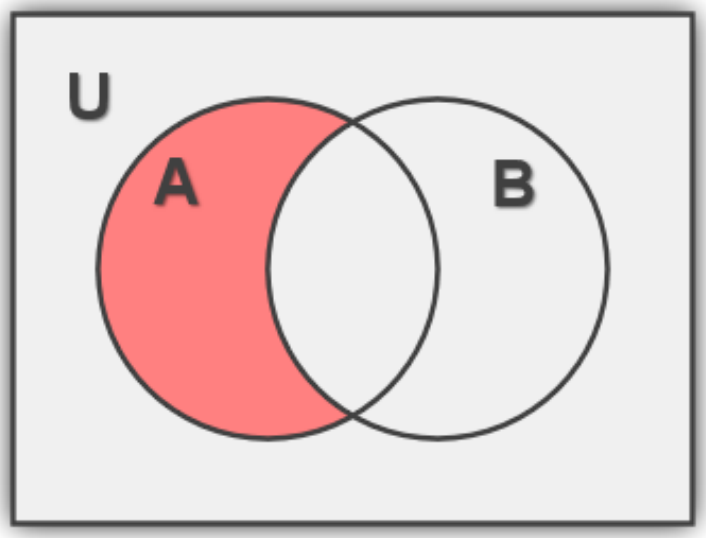
```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'a', 'r', 'b', 'c', 'd'}
```

> Sets contain unique elements, therefore, even if duplicate elements are provided they will be removed.

Set Operators

Difference

The difference between two sets are the elements that are unique to the set to the left of the `-` operator, with duplicates removed.

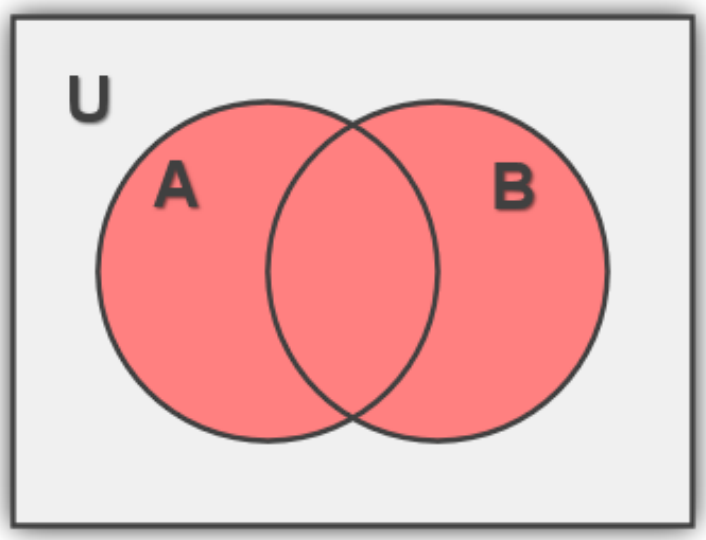



```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a - b
{'r', 'd', 'b'}
```

> This results the letters that are in a but not in b

Union

The union between two sets is a sequence of the all the elements of the first and second sets combined, with duplicates removed.

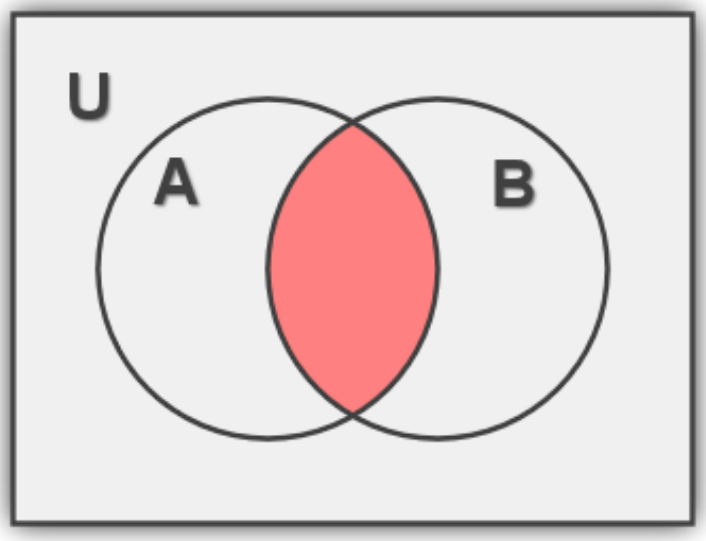



```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a | b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
```

> This returns letters that are in a or b both

Intersection

The intersection between two sets is a sequence of the elements which are in both sets, with duplicates removed.

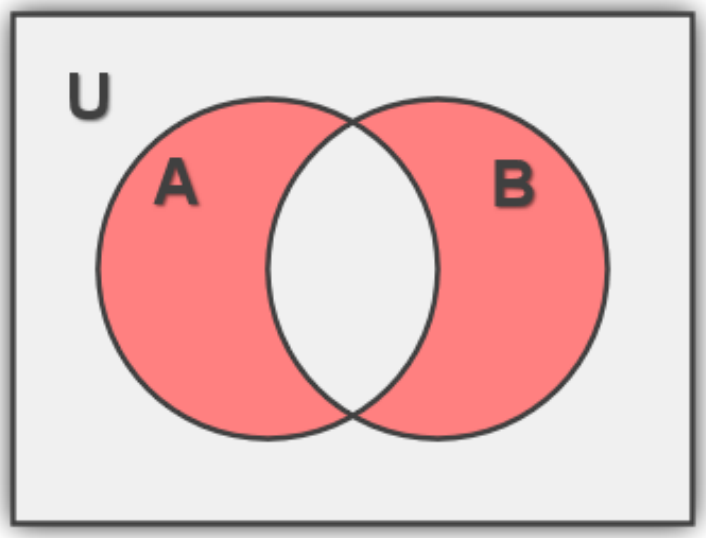


```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a & b
{'a', 'c'}
```

> This returns letters that are in both a and b

Symmetric Difference

The symmetric difference is the elements that are only in the first set plus the elements that are only in the second set, with duplicates removed.




```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a ^ b
{'r', 'd', 'b', 'm', 'z', 'l'}
```

> This returns the letters that are in a or b but not in both (also known as exclusive or)

Set Functions

Function	Description
`all()`	returns True if all elements of the set are true (or if the set is empty).
`any()`	returns True if any element of the set is true. If the set is empty, return False.
`enumerate()`	returns an enumerate object. It contains the index and value of all the items of set as a pair.
`len()`	returns the number of items in the set.
`max()`	returns the largest item in the set.
`min()`	returns the smallest item in the set.
`sorted()`	returns a new sorted list from elements in the set (does not alter the original set).
`sum()`	returns the sum of all elements in the set.

Set Methods

Method	Description
`set.add(new)`	adds a new element
`set.clear()`	remove all elements
`set.copy()`	returns a shallow copy of a set
`set.difference(set2)`	returns the difference of set and set2
`set.difference_update(set2)`	removes all elements of another set from this set
`set.discard(element)`	removes an element from set if it is found in set. (Do nothing if the element is not in set)
`set.intersection(sets)`	return the intersection of set and the other provided sets
`set.intersection_update(sets)`	updates set with the intersection of set and the other provided sets
`set.isdisjoint(set2)`	returns True if set and set2 have no intersection
`set.issubset(set2)`	returns True if set2 contains set
`set.issuperset(set2)`	returns True if set contains set2
`set.pop()`	removes and returns an arbitrary element of set.
`set.remove(element)`	removes element from a set.
`set.symmetric_difference(set2)`	returns the symmetric difference of set and set2
`set.symmetric_difference_update(set2)`	updates set with the symmetric difference of set and set2
`set.union(sets)`	returns the union of set and the other provided sets
`set.update(set2)`	update set with the union of set and set2

Build a dictionary of NT counts using a set and loops

Let us put a twist on our nt count script. Let's use a set to find all the unique nts, then use the string `count()` method to count the nucleotide instead if incrementing the count as we did earlier.

Code:

```
#!/usr/bin/env python3
```

```
# create a new empty dictionary
nt_count = {}

# get a set of unique characters in our DNA string

dna = 'GTACCNTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
unique = set(dna)

print('unique nt: ', unique) ## {'C', 'A', 'G', 'T', 'N'}

# iterate through each unique nucleotide
for nt in unique:
  # count the number of this unique nt in dna
  count = dna.count(nt)

  # add our count to our dict
  nt_count[nt] = count


print('nt count:', nt_count)
```

Output:

```
unique nt:  {'N', 'C', 'T', 'G', 'A'}
nt count: {'G': 20, 'T': 21, 'A': 13, 'C': 16, 'N': 1}
```

> We have the count for all NTs even ones we might not expect.

Link to Python 5 Problem Set

Python 6

I/O and Files

I/O stands for input/output. The in and out refer to getting data into and out of your script. It might be a little surprising at first, but writing to the screen, reading from the keyboard, reading from a file, and writing to a file are all examples of I/O.

Writing to the Screen

You should be well versed in writing to the screen. We have been using the `print()` function to do this.

```
>>> print ("Hello, PFB!")
Hello, PFB!
```

> Remember this example from one of our first lessons?

Reading input from the keyboard

This is something new. There is a function which prints a message to the screen and waits for input from the keyboard. This input can be stored in a variable. It always starts as a string. Convert to an int or float if you want a number. When you are done entering text, press the enter key to end the input. A newline character is not included in the input.

```
>>> user_input = input("Type Something Now: ")
Type Something Now: Hi
>>> print(user_input)
Hi
>>> in_str = input("Enter a number: ")
>>> type(in_str)
<class 'str'>
>>> num = int(in_str)
>>> num
445
```

Reading from a File

Mostly you will read data from files.

The first thing to do with a file is open it. We can do this with the `open()` function. The `open()` function takes the file name and access mode as arguments and returns a file object.

The most common access modes are read (r) and write (w).

Open a File

```
>>> seq_file_obj = open("seq.nt.txt","r")
```

> `seq_file_obj` is a name of a variable. This can be anything, but make it a helpful name that describes what kind of file you are opening and to distinguish it from the filename.

Reading the contents of a file

Now that we have opened a file and created a file object we can do things with it, like read from the file. Let's read all the contents at once.

Before we do that, let's go to the command line and `cat` the contents of the file to see what's in it first.

```
$ cat seq.nt.txt
ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAG
ACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG
$
```

Note the new lines. Now, lets print the contents to the screen with Python. We will use `read()` to read the entire contents of the file into a variable. You don't usually use this function (see below).

```
>>> seq_file_obj = open("seq.nt.txt","r")
>>> contents = seq_file_obj.read()
>>> print(contents)  # note newline characters are part of the file!
ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAG
ACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG

>>> seq_file_obj.close()
```

> The complete contents can be retrieved with the `read()` method. Notice the newlines are maintained when `contents` is printed to the screen. `print()` adds another new line when it is finished printing. It is good practice to close your file. Use the `close()` method.

Here's another way to read data in from a file. A `for` loop can be used to iterate through the file one line at a time.

```
#!/usr/bin/env python3

seq_file_obj = open("seq.nt.txt","r")
for line in seq_file_obj: # Python magic: reads in a line from file
  print(line)
seq_file_obj.close()
```

Output:

```
$ python3 file_for.py
ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAG

ACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG
```

> Notice the blank line at after each line we print. `print()` adds a newline and we have a newline at the end of each line in our file. Use `rstrip()` method to remove the newline from each line.

Let's use `rstrip()` method to remove the newline from our file input.

```
$ cat file_for_rstrip.py
#!/usr/bin/env python3

seq_file_obj = open("seq.nt.txt","r")
for line in seq_file_obj:
  line = line.rstrip()
  print(line)
seq_file_obj.close()
```

> `rstrip()` without any parameters returns a string with whitespace removed from the end.

Output:

```
$ python3 file_for_rstrip.py
ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAG
ACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG
```

> Where do the newlines in the above output come from?

Opening a file with `with open() as fh:`

Many people add this, because it closes the file for you automatically. Good programming practice. Your code will clean up as it runs. For more advanced coding, `with ... as ...` saves limited resources like filehandles and database connections. For now, we just need to know that the `with ... as ...:` does the same as `fh = open(...) ... fh.close()`. So here's what the adapted code looks like

```
#!/usr/bin/env python3

with open("seq.nt.txt","r") as seq_file_obj: #cleans up after exiting
                                             # the 'with' block
  for line in seq_file_obj:
    line = line.rstrip()
    print(line)
# file gets closed for you here.
```

Writing to a File

Writing to a file just requires opening a file for writing then using the `write()` method.

The `write()` method is like the `print()` function. The biggest difference is that it writes to your file object instead of the screen. Unlike `print()`, it does not add a newline by default. `write()` takes a single string argument.

Let's write a few lines to a file named "writing.txt".

```
#!/usr/bin/env python3

fo = open("writing.txt" , "w")  # note that we are writing so the mode is "w"
fo.write("One line.\n")
fo.write("2nd line.\n")
fo.write("3rd line" + " has extra text\n")
some_var = 5
fo.write("4th line has " + str(some_var) + " words\n")  # the write() method does not convert
ints for you
fo.close()
print("Wrote to file 'writing.txt'") # it's nice to tell the user you wrote a file
```

Output:

```
$ python3 file_write.py
Wrote to file 'writing.txt'
$ cat writing.txt
One line.
2nd line.
3rd line has extra text
4th line has 5 words
```

Now, let's get crazy! Lets read from one file a line at a time. Do something to each line and write the results to a new file.

```
#!/usr/bin/env python3

total_nts = 0
# open two file objects, one for reading, one for writing
with open("seq.nt.txt","r") as seq_read, open("nt.counts.txt","w") as seq_write:
  for line in seq_read:
    line = line.rstrip()
    nt_count = len(line)
    total_nts += nt_count
    seq_write.write(f"{nt_count}\n")

  seq_write.write(f"Total: {total_nts}\n") # better than concatenation + str()
```

```
print("Wrote 'nt.counts.txt'")
```

Output:

```
$ python3 file_read_write.py
Wrote 'nt.counts.txt'
$ cat nt.counts.txt
71
71
Total: 142
```

> The file we are reading from is named, "seq.nt.txt"
> The file we are writing to is named, "nt.counts.txt"
> We read each line, calculate the length of each line, and print the length
> We also create a variable to keep track of the total nt count
> At the end, we print out the total count of nts
> Finally, we close each of the files

Building a Dictionary from a File

This is a very common task. It will use a loop, file I/O, and a dictionary.

Assume we have a file called "sequence_data.txt" that contains tab-delimited gene names and sequences that looks something like this

```
TP53    GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
BRCA1   GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA
```

How can we read this whole file in to a dictionary?

```
#!/usr/bin/env python3

genes = {}
with open("sequence_data.txt","r") as seq_read:
  for line in seq_read:
    line = line.rstrip()
    gene_id,seq = line.split() #split on whitespace
    genes[gene_id] = seq
print(genes)
```

Output:

```
{'TP53': 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC', 'BRCA1':
'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'}
```

Link to Python 6 Problem Set