# Python 4

## 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 the while loop block as long as a given condition returns True.

### While Loop Syntax

```
while expression:
  statement(s)
```

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

Code:

```python
#!/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

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

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

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

Output:

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

> What caused this code 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 forever printing use Cntl+C.

## While/Else

An Else statment can be used with a while statement. It behaves in the same way as and Else with an If statement. When the while statement is false, the else block is excuted ONE TIME.

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

count = 0
while count < 5:
  print("count:" , count)
  count+=1
else:
  print("count:", count, "is now not less than 5")
print("Done")
```

Output:

```
$ python while_else.py
count: 0
count: 1
count: 2
count: 3
count: 4
count: 5 is now not less than 5
Done
```

> The while loop was executed five times like before. Now when count is equal to 5 and therefore not less than 5, the else block is executed. Finally the lines of code outside the while/else are executed.

### 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

```python
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:

```python
#!/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 is next example is using a for loop to iterating 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:

```python
#!/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.

```python
>>> 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:

```python
#!/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
```

## For/Else

An else statement can be used with a for loop as well. The else block of code will be executed when the for loop exits normally.

Code:

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

for num in range(5):
  print(num)
else:
  print("Completed for loop")
```

Output:

```
$ python3 list_range_else.py
0
1
2
3
4
Completed for loop
```

## Loop Control

Loops 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 inital condition (count < 5) is still True.

## Loop Control: Continue

Code:

```python
#!/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 iterable, 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.

### 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]
```

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 mathatical 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 ]

## Dictionaries

Dictionaries are another iterable, like a string and list. Unlike strings and lists, dictionaries are not a sequence, or in other words, they are unordered and the position is not important.

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 assess 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)
{'BRCA1': 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA', 'TP53': 'atg'}
```

> 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)
{'BRCA1': 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA', 'TP53': 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTCTAGAGCCACCGTCCAGGGAGCAGGTAGCTGCTGGGCTCCGGGGACACTTTGCGTTCGGGCTGGGAGCGTG'}
```

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

## Accessing Each Dictionary Key/Value

Since a dictionary is a sequence 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
>>>
```

## Sorting Dictionary Keys

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

```
for key in sorted(genes):
  print(key, '=>' , genes[key])
```

> This will print the same order of keys every time you run your script.

## 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 dataypes 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 only matters in multidementional datastructures. |
| `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.keys()` | Returns list of keys |
| `dict.get(key, default = None)` | get value from dict[key], use default if not 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 |

## 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

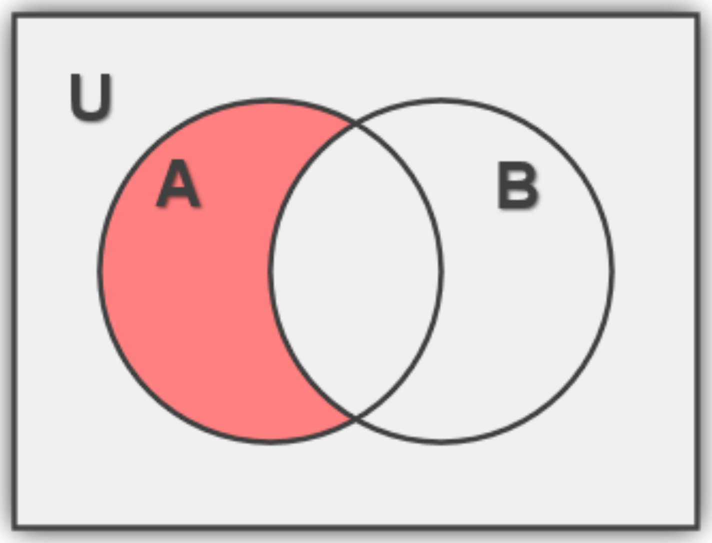
```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'a', 'r', 'b', 'c', 'd'}
```

> 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.

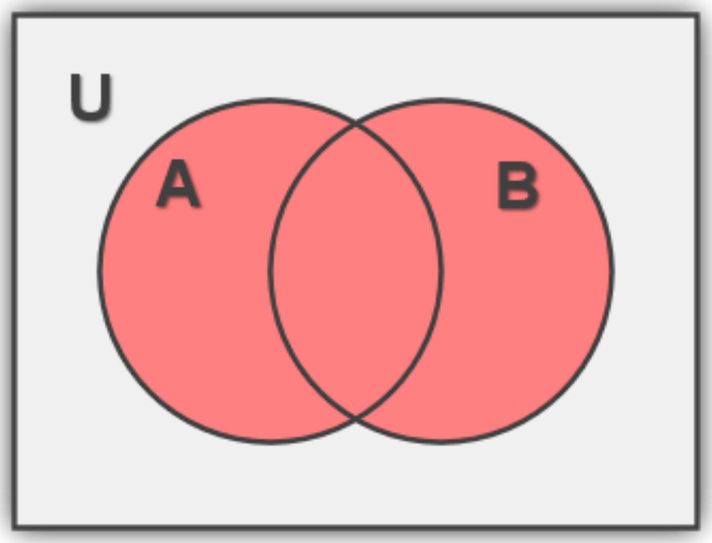


```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a - b
{'r', 'd', 'b'}
```

> 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.

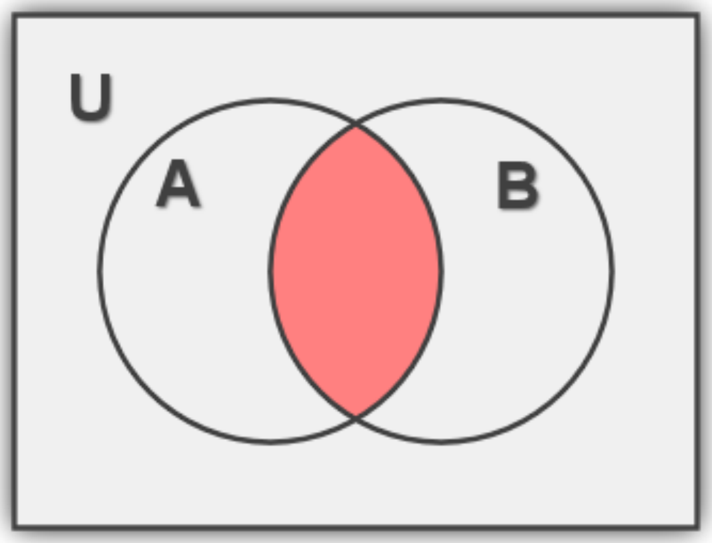


```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a | b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
```

> 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.

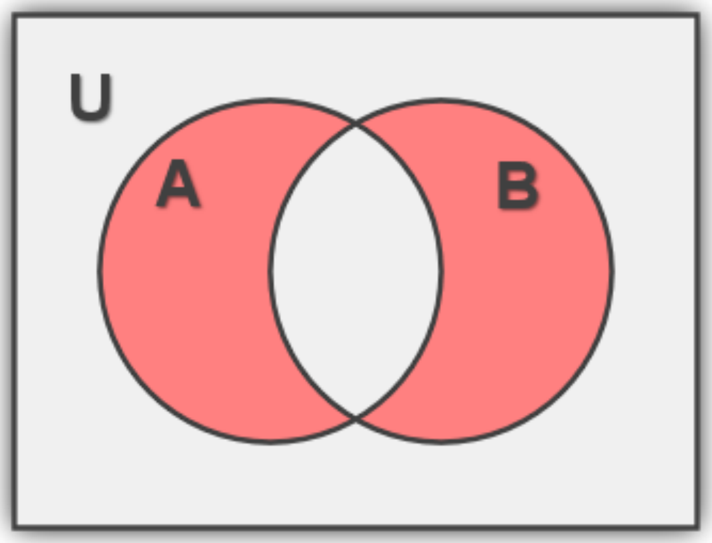


```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a & b
{'a', 'c'}
```

> This returns letters that are in both a and b

#### Symmetric Difference

The symetric difference are 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 arbitary 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 |

## Link to Python 4 Problem Set

# Python 5

## 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, PFB2017!")
Hello, PFB2017!
```

> 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.

```
>>> user_input = input("Type Something Now: ")
Type Something Now: Hi
>>> print(user_input)
Hi
```

> All inputed text will be treated as a string. If you are to do math with the input, convert to an int or float first.

### Reading from a File

Most of the data we will be dealing with will be contained in 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, access mode as arguments and returns a file object.

The most common access modes are reading (r) and writing (w).

### Open a File

```
>>> file_object = open("seq.nt.fa","r")
```

> 'file_object' is a name of a variable. This can be anything.