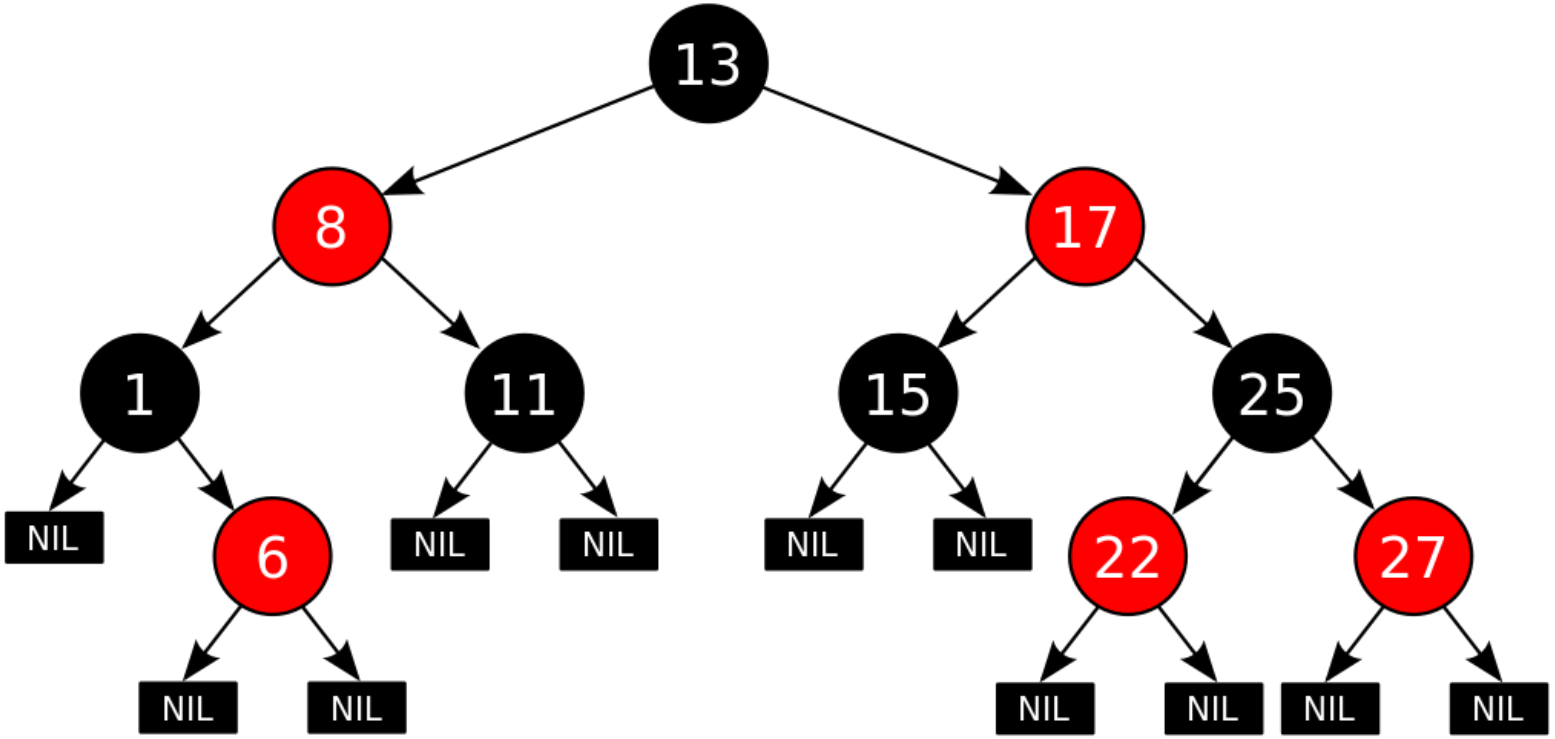
Data Structures

Python for Genomic Data Science



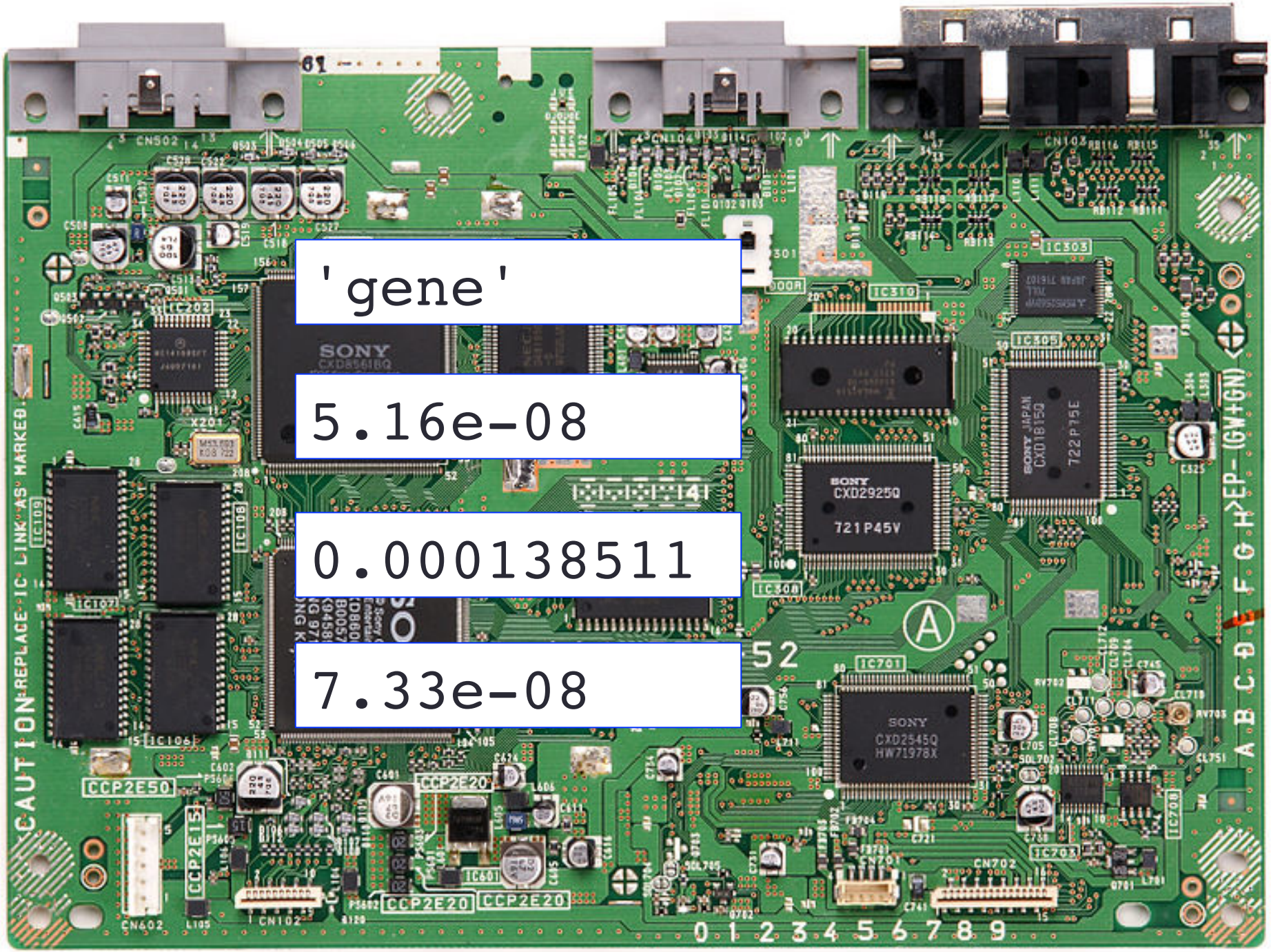
Lists

A *list* is an ordered set of values:

```
['gene', 5.16e-08, 0.000138511, 7.33e-08]
```

You can create a variable to hold this list:

```
>>> gene_expression=['gene',5.16e-08, 0.000138511, 7.33e-08]
```


'gene'
5.16e-08
0.000138511
7.33e-08
SONY
CXD2925Q
721P45V
SONY
CXD2545Q
HW71978X
CAUTION-REPLACE-IC LINK"AS" MARKED.
0 1 2 3 4 5 6 7 8 9

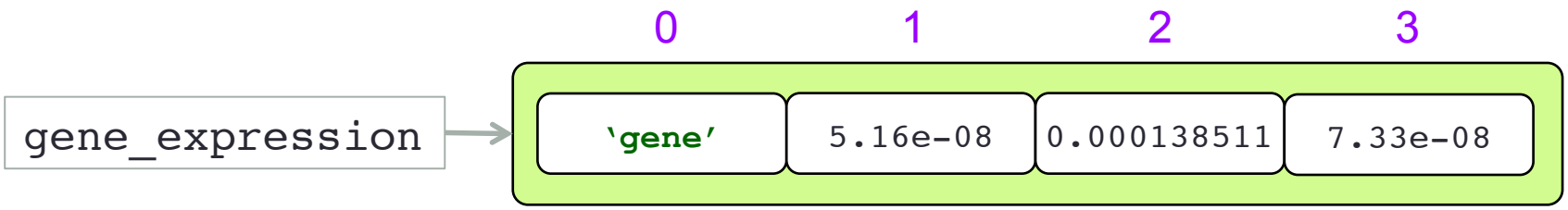
Lists

A *list* is an ordered set of values:

```
['gene', 5.16e-08, 0.000138511, 7.33e-08]
```

You can create a variable to hold this list:

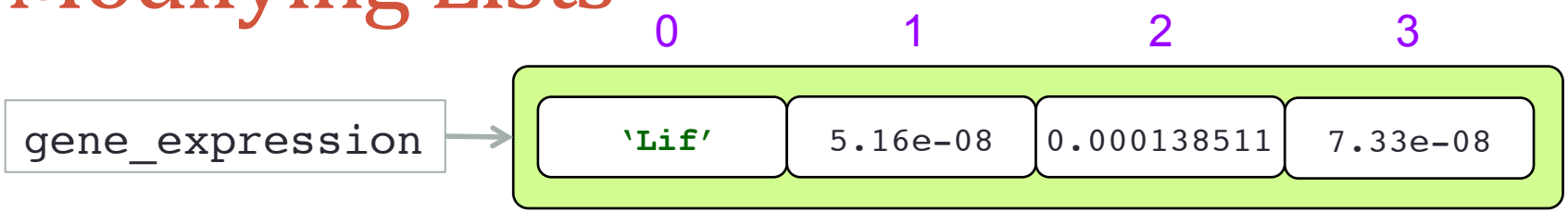
```
>>> gene_expression=['gene',5.16e-08, 0.000138511, 7.33e-08]
```



You can access individual list elements:

```
>>> print(gene_expression[2])
0.000138511
>>> print(gene_expression[-1])
7.33e-08
```

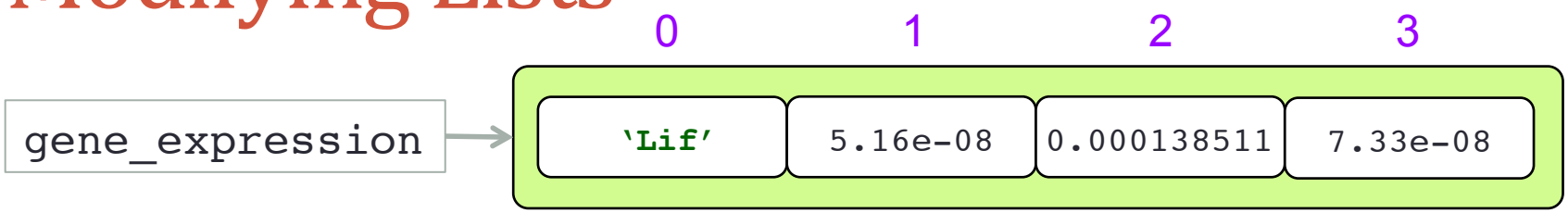
Modifying Lists



You can change an individual list element:

```
>>> gene_expression[0]='Lif'
>>> print(gene_expression)
['Lif', 5.16e-08, 0.000138511, 7.33e-08]
```

Modifying Lists



You can change an individual list element:

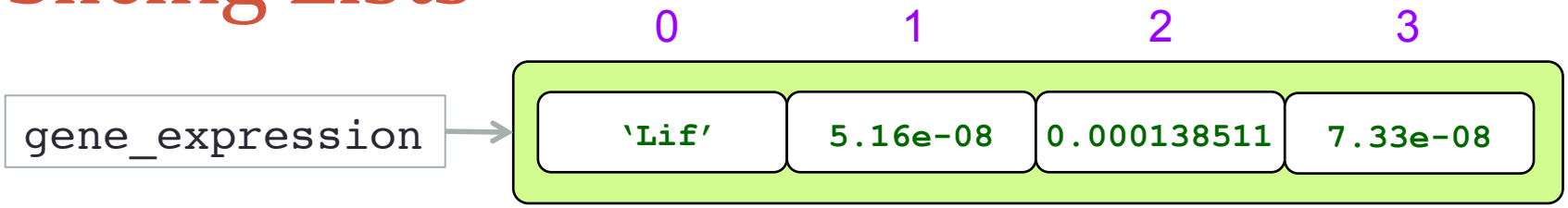
```
>>> gene_expression[0]='Lif'
>>> print(gene_expression)
['Lif', 5.16e-08, 0.000138511, 7.33e-08]
```

Don't change an element in a string!

```
>>> motif ='nacggggtc'
>>> motif[0]='a'
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    motif[0]='a'
TypeError: 'str' object does not support item assignment
```

Unlike strings, which are *immutable*, lists are a *mutable* type!

Slicing Lists



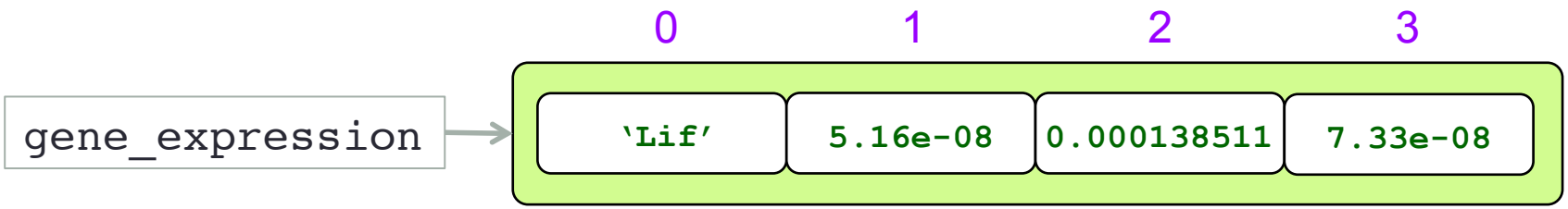
You can slice a list (it will create a new list):

```
>>> gene_expression[-3:]
[5.16e-08, 0.000138511, 7.33e-08]
```

The following special slice returns a new copy of the list:

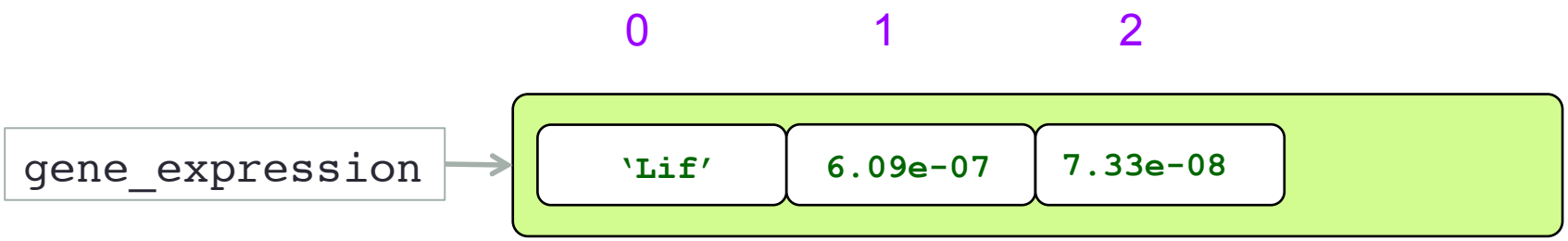
```
>>> gene_expression[:]
['Lif', 5.16e-08, 0.000138511, 7.33e-08]
```

Slicing Lists



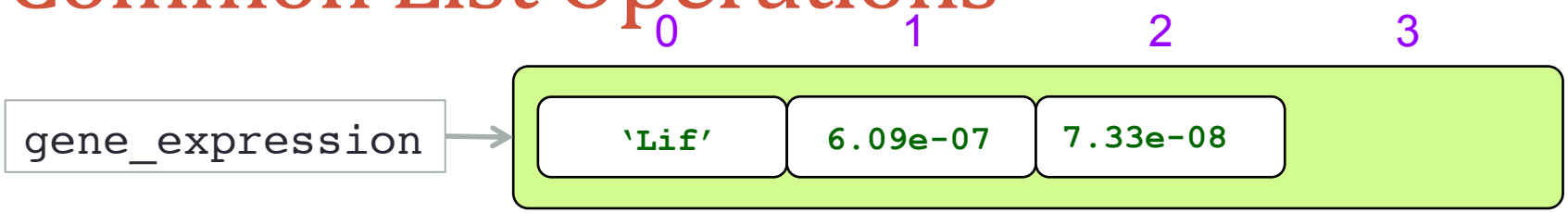
Assignment to slices is also possible, and this can change the list:

```
>>> gene_expression[1:3]=[6.09e-07]
```




```
>>> gene_expression[:]=[] # this clears the list
```

Common List Operations



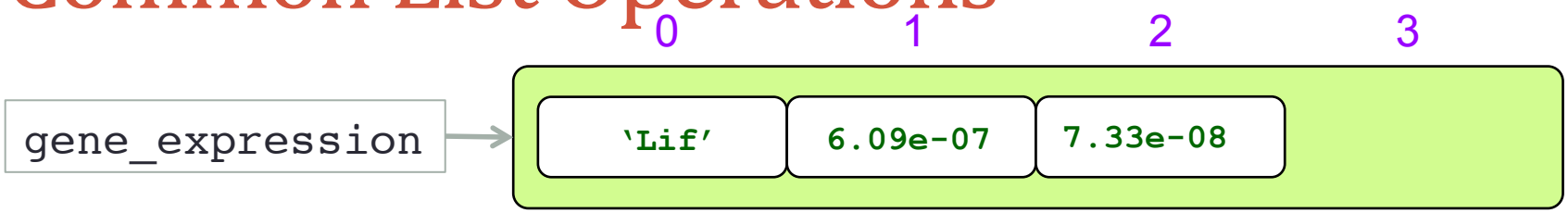
Like strings, lists also support concatenation:

```
>>> gene_expression+[5.16e-08, 0.000138511]
['Lif', 6.09e-07, 7.33e-08, 5.16e-08, 0.000138511]
```

The built-in function `len()` also applies to lists:

```
>>> len(gene_expression)
3
```

Common List Operations



Like strings, lists also support concatenation:

```
>>> gene_expression+[5.16e-08, 0.000138511]
['Lif', 6.09e-07, 7.33e-08, 5.16e-08, 0.000138511]
```

The built-in function `len()` also applies to lists:

```
>>> len(gene_expression)
3
```

The `del` statement can be used to remove elements and slices from a list destructively:

```
>>> del gene_expression[1]
>>> gene_expression
['Lif', 7.33e-08]
```

Lists As Objects

The list data type has several methods. Among them:

- a method to extend a list by appending all the items in a given list:

```
>>> gene_expression.extend([5.16e-08, 0.000138511])
>>> gene_expression
['Lif', 7.33e-08, 5.16e-08, 0.000138511]
```

Lists As Objects

The list data type has several methods. Among them:

- a method to extend a list by appending all the items in a given list:

```
>>> gene_expression.extend([5.16e-08, 0.000138511])
>>> gene_expression
['Lif', 7.33e-08, 5.16e-08, 0.000138511]
```

- a method to count the number of times an element appears in a list:

```
>>> print(gene_expression.count('Lif'),gene_expression.count('gene'))
1 0
```

- a method to reverse all elements in a list:

```
>>> gene_expression.reverse()
>>> gene_expression
[0.000138511, 5.16e-08, 7.33e-08, 'Lif']
```

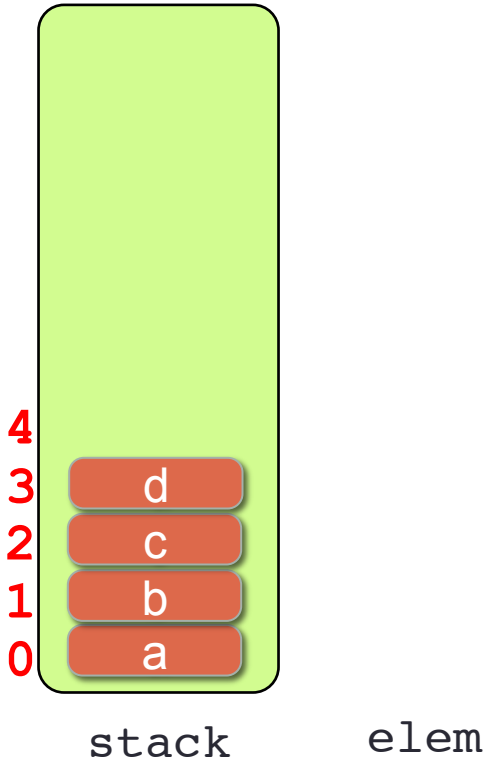
You can find all the methods of the list object using the `help()` function:

```
>>> help(list)
```

Lists As Stacks

The list methods `append` and `pop` make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out").

```
>>> stack=['a','b','c','d']
```



Lists As Stacks

The list methods `append` and `pop` make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out").

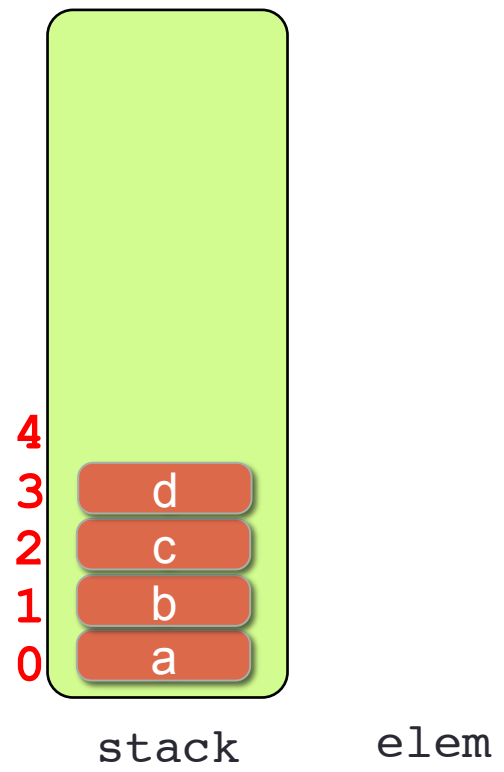
```
>>> stack=['a','b','c','d']
```

To add an item to the top of the stack, use `append()`:

```
>>> stack.append('e')
```

To retrieve an item from the top of the stack, use `pop()`:

```
>>> elem=stack.pop()
>>> elem
'e'
```



Sorting Lists

There are two ways to sort lists:

- one way uses the `sorted()` built-in function:

```
>>> mylist=[3,31,123,1,5]
>>> sorted(mylist)
[1, 3, 5, 31, 123]
>>> mylist
[3, 31, 123, 1, 5]
```

- another way is to use the list `sort()` method:

```
>>> mylist.sort()
```

Sorting Lists

There are two ways to sort lists:

- one way uses the `sorted()` built-in function:

```
>>> mylist=[3,31,123,1,5]
>>> sorted(mylist)
[1, 3, 5, 31, 123]
>>> mylist
[3, 31, 123, 1, 5]
```

- another way is to use the list `sort()` method:

```
>>> mylist.sort()
>>> mylist
[1, 3, 5, 31, 123]
```

the `sort()` method modifies the list!

The elements of the list don't need to be numbers:

```
>>> mylist=['c','g','T','a','A']
>>> print(sorted(mylist))
['A', 'T', 'a', 'c', 'g']
```

Tuples

A *tuple* consists of a number of values separated by commas, and is another standard sequence data type, like strings and lists.

```
>>> t=1,2,3
>>> t
(1, 2, 3)
>>> t=(1,2,3)
>>> t
(1, 2, 3)
```

We may input tuples may with or without surrounding parentheses.

Tuples

A *tuple* consists of a number of values separated by commas, and is another standard sequence data type, like strings and lists.

```
>>> t=1,2,3
>>> t
(1, 2, 3)
>>> t=(1,2,3)
>>> t
(1, 2, 3)
```

We may input tuples may with or without surrounding parentheses.

Tuples have many common properties with lists, such as indexing and slicing operations, but while lists are mutable, tuples are immutable, and usually contain an heterogeneous sequence of elements.

Sets

A *set* is an unordered collection with no duplicate elements. Set objects support mathematical operations like *union*, *intersection*, and *difference*.

```
>>> brca1={'DNA repair','zinc ion binding','DNA
binding','ubiquitin-protein transferase activity', 'DNA
repair','protein ubiquitination'}
>>> brca1
{'DNA repair','zinc ion binding','DNA binding','ubiquitin-protein
transferase activity', 'DNA repair','protein ubiquitination'}
```

Sets

A *set* is an unordered collection with no duplicate elements. Set objects support mathematical operations like *union*, *intersection*, and *difference*.

```
>>> brca1={'DNA repair','zinc ion binding','DNA
binding','ubiquitin-protein transferase activity', 'DNA
repair','protein ubiquitination'}
>>> brca1
{'DNA repair','zinc ion binding','DNA binding','ubiquitin-protein
transferase activity’,'protein ubiquitination'}

>>> brca2={'protein binding','H4 histone acetyltransferase
activity','nucleoplasm', 'DNA repair','double-strand break
repair', 'double-strand break repair via homologous
recombination'}
```

Operation with Sets

```
>>> brca1 | brca2
{'DNA repair','zinc ion binding','DNA
binding','ubiquitin-protein transferase
activity’,'protein ubiquitination', 'protein
binding','H4 histone acetyltransferase
activity','nucleoplasm','double-strand break repair',
'double-strand break repair via homologous
recombination'}
```

union

```
>>> brca1 & brca2
{'DNA repair'}
```

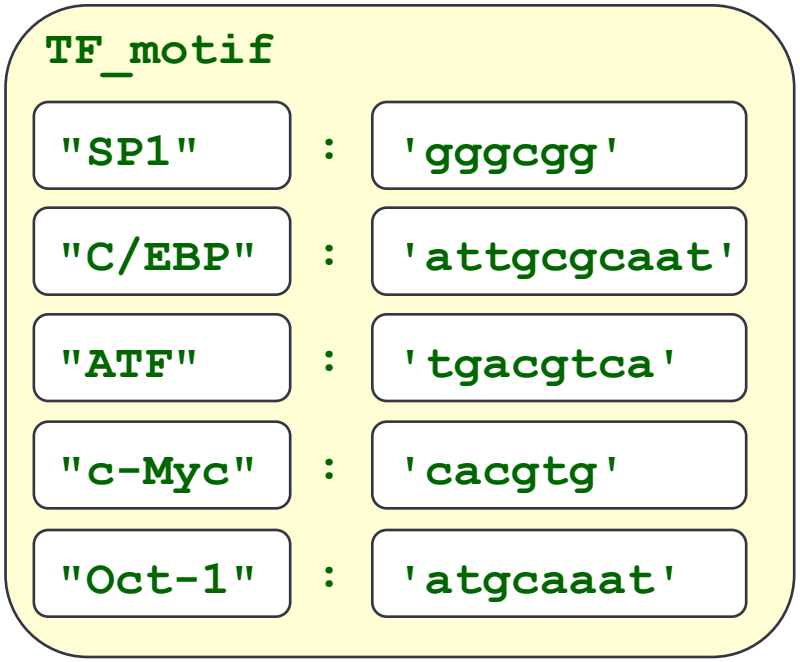
intersection

```
>>> brca1 – brca2
{'zinc ion binding','DNA binding','ubiquitin-protein
transferase activity’,'protein ubiquitination'}
```

difference

Dictionaries

A *dictionary* is an unordered set of *key* and *value* pairs, with the requirement that the keys are unique (within one dictionary).



keys: can be any immutable type: e.g. strings, numbers.

values: can be any type.

```
>>> TF_motif =
{'SP1' :'gggcgg',
'C/EBP':'attgcgcaat',
'ATF':'tgacgtca',
'c-Myc':'cacgtg',
'Oct-1':'atgcaaat'}
```

Each key is separated from its value by a colon.

Accessing Values From A Dictionary

Use a dictionary key within square brackets to obtain its value:

```
>>> TF_motif={'SP1' : 'gggcgg', 'C/EBP':'attgcgcaat',
'ATF':'tgacgtca','c-Myc':'cacgtg','Oct-1':'atgcaaat'}
>>> print("The recognition sequence for the ATF transcription
is %s." % TF_motif['ATF'])
The recognition sequence for the ATF transcription is
tgacgtca.
```

Attempting to access a key that is not part of the dictionary produces an error:

```
>>> print("The recognition sequence for the NF-1 transcription
is %s." % TF_motif['NF-1'])
Traceback (most recent call last):
  File "<pyshell#291>", line 1, in <module>
    print("The recognition sequence for the ATF transcription
is %s"%TF_motif['NF-1'])
KeyError: 'NF-1'
```

Check first if a key is present!

```
>>> 'NF-1' in TF_motif
False
```

Updating A Dictionary

```
>>> TF_motif={'SP1' : 'gggcgg', 'C/EBP':'attgcgcaat',
'ATF':'tgacgtca','c-Myc':'cacgtg'}
```

- Add a new *key:value* pair to the dictionary:

```
>>> TF_motif['AP-1']='tgagtca'
>>> TF_motif
{'ATF': 'tgacgtca', 'c-Myc': 'cacgtg', 'SP1': 'gggcgg',
'C/EBP': 'attgcgcaat', 'AP-1': 'tgagtca'}
```

- Modify an existing entry:

```
>>> TF_motif['AP-1']='tga(g/c)tca'
>>> TF_motif
{'ATF': 'tgacgtca', 'c-Myc': 'cacgtg', 'SP1': 'gggcgg',
'C/EBP': 'attgcgcaat', 'AP-1': 'tga(g/c)tca’}
```

Updating A Dictionary (cont'd)

```
>>> TF_motif
{'ATF': 'tgacgtca', 'c-Myc': 'cacgtg', 'SP1': 'gggcgg', 'C/
EBP': 'attgcgcaat', 'AP-1': 'tga(g/c)tca'}
```

- Delete a key from the dictionary:

```
>>> del TF_motif['SP1']
>>> TF_motif
{'ATF': 'tgacgtca', 'c-Myc': 'cacgtg', 'C/EBP': 'attgcgcaat',
'AP-1': 'tga(g/c)tca'}
```

- Add another dictionary (multiple *key:value* pairs) to the current one:

Note the overlap with the current dictionary.

```
>>> TF_motif.update({'SP1': 'gggcgg', 'C/EBP': 'attgcgcaat',
'Oct-1': 'atgcaaa'})
>>> TF_motif
{'ATF': 'tgacgtca', 'c-Myc': 'cacgtg', 'SP1': 'gggcgg', 'C/
EBP': 'attgcgcaat', 'Oct-1': 'atgcaaa', 'AP-1': 'tga(g/c)tca'}
```

Listing All Elements In A Dictionary

- The size of a dictionary can be easily obtained by using the built-in function len():

```
>>> len(TF_motif)
6
```

- It is possible to get a <u>list</u> of all the *keys* in the dictionary:

```
>>> list(TF_motif.keys())
['ATF', 'c-Myc', 'SP1', 'C/EBP', 'Oct-1', 'AP-1']
```

- Similarly you can get a list of all the *values:*

```
>>> list(TF_motif.values())
['tgacgtca', 'cacgtg', 'gggcgg', 'attgcgcaat', 'atgcaaa',
'tga(g/c)tca']
```

- The lists found as above are in arbitrary order, but if you want them sorted you can use the `sorted()` function:

```
>>> sorted(TF_motif.keys())
['AP-1', 'ATF', 'C/EBP', 'Oct-1', 'SP1', 'c-Myc']
>>> sorted(TF_motif.values())
['atgcaaa', 'attgcgcaat', 'cacgtg', 'gggcgg', 'tga(g/c)tca',
'tgacgtca']
```

Sequence Data Types Comparison

Action	Strings	Lists	Dictionaries
Creation	"...", '...', "'...'"	[a, b, ..., n]	{keya: a, keyb: b, ..., keyn:n }
Access to an element	s[i]	L[i]	D[key]
Membership	c in s	e in L	key in D
Remove en element	Not Possible s = s[:i−1]+s[i+1:]	del L[i]	del D[key]
Change an element	Not Possible s=s[:i−1]+new+s[i+1:]	L[i]=new	D[key]=new
Add an element	Not Possible s=s + new	L.append(e)	D[newkey]=val
Remove consecutive elements	Not Possible s=s[:i]+s[k:]	del L[i:k]	Not Possible, not ordered but, remove all D.clear()
Change consecutive elements	Not Possible s=s[:i]+news+s[k:]	L[i:k]=Lnew	Not Possible
Add more than one element	Not Possible s=s+news	L.extend(newL) or L = L + Lnew	D.update(newD)

Adapted from http://www.pasteur.fr/formation/infobio/python/ch09.html