Data structures Part 1: Lists

Lesson 4: 9/14/2017

Speaker: Samantha Klasfeld

Today's Schedule

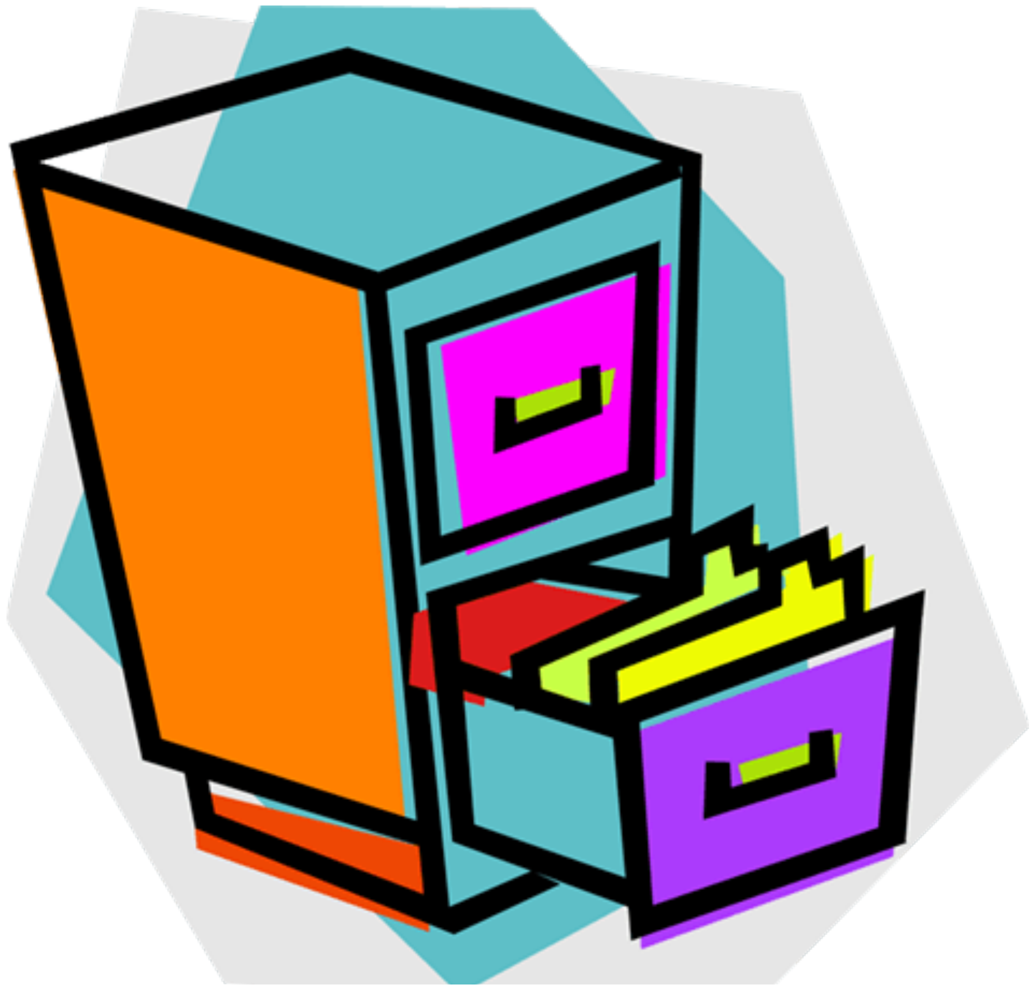
- Introduction to Data Structures
- Lists
- File parsing with `.split()`

1. Introduction to Data Structures

What is a Data Strucutre?

- A data structure is an object for storing large amounts of data (numbers, strings, etc) in an organized manner, making storage and retrieval easier

It may be helpful to imagine a data structure is a file cabinet…



Each object can be the contents of each folder within the cabinet

2. Lists

What is a list?

- A **list** is a built-in data structure in Python (along with sets, tuples, and dictionaries) that is ordered by indexes

- Python lists can look like this:

```
["hello", "world", "i", "am", "sam"]
```

OR

```
[5, 4, 2, 1, 1]
```

OR

```
[5.001, 4.4, 2.0, .991, 42.6]
```

OR

```
[True, False, False]
```

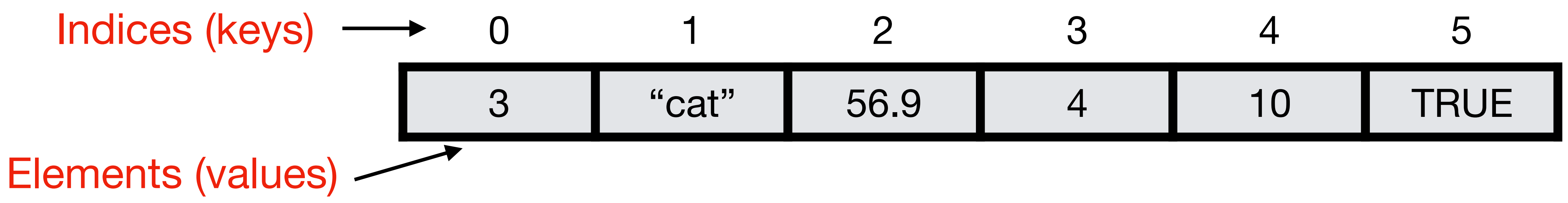
OR

```
[]
```

OR

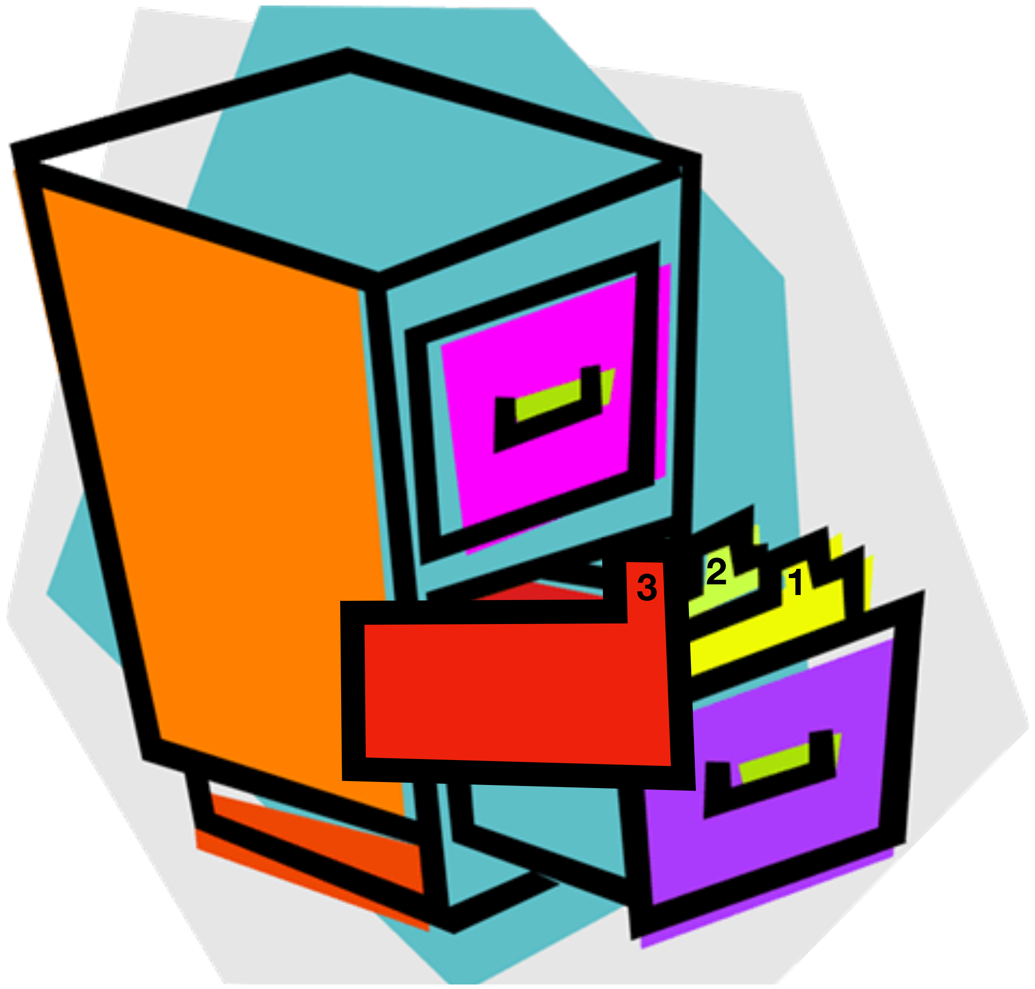
```
[3, "cat", 56.9, 4, 10, True]
```

- However, it may be more helpful to think of them like this



where each **element** is given an index, starting at 0.

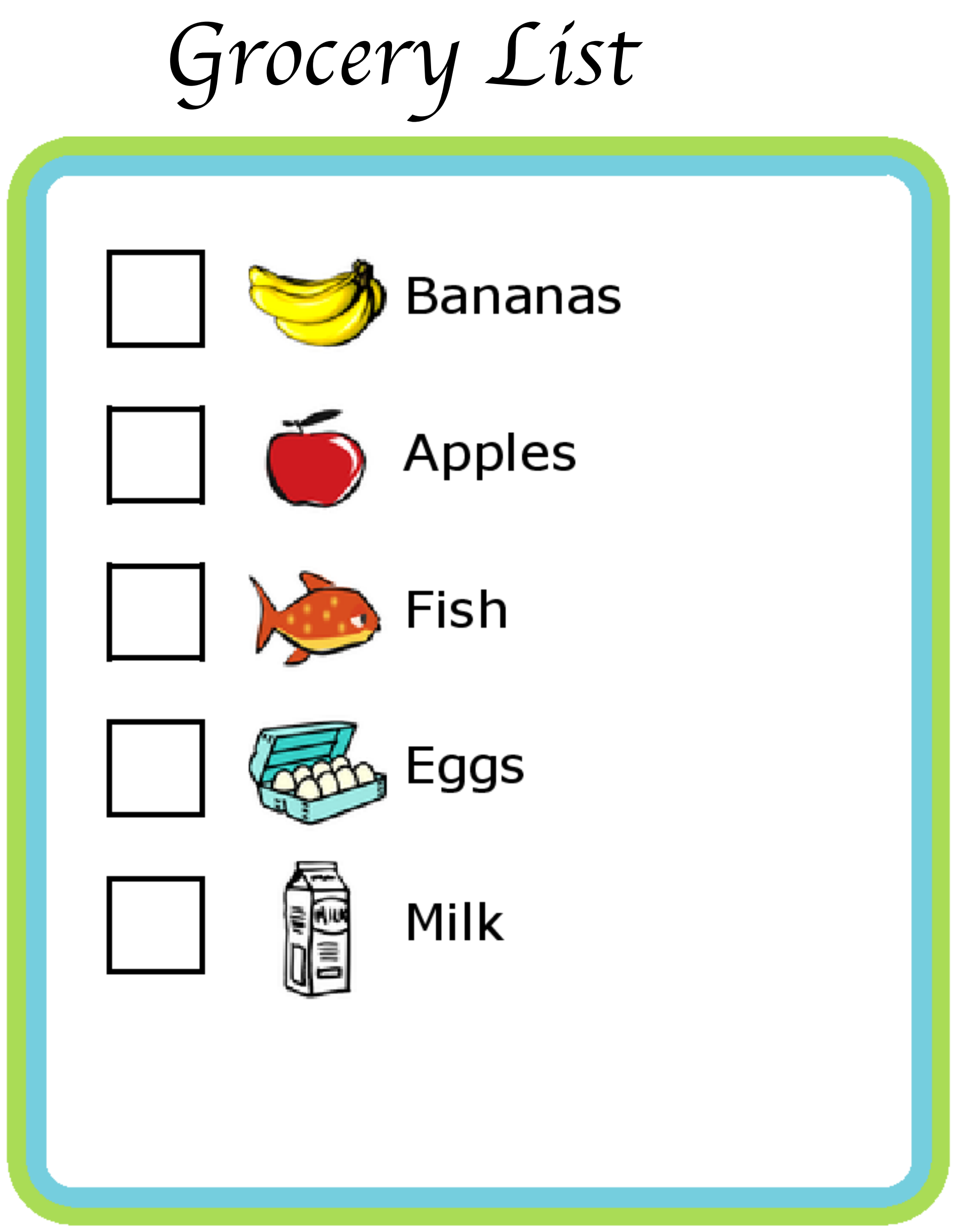
If you think of lists as a type of file cabinet, then the indices are the labels on each folder



56.9

the elements are the contents inside the folders

Accessing Elements in a List



Here is my grocery list.

- What is the third item on my grocery list?
- What are the first 2 items on my grocery list?
- What are the last two items on my grocery list?

Accessing elements in a list

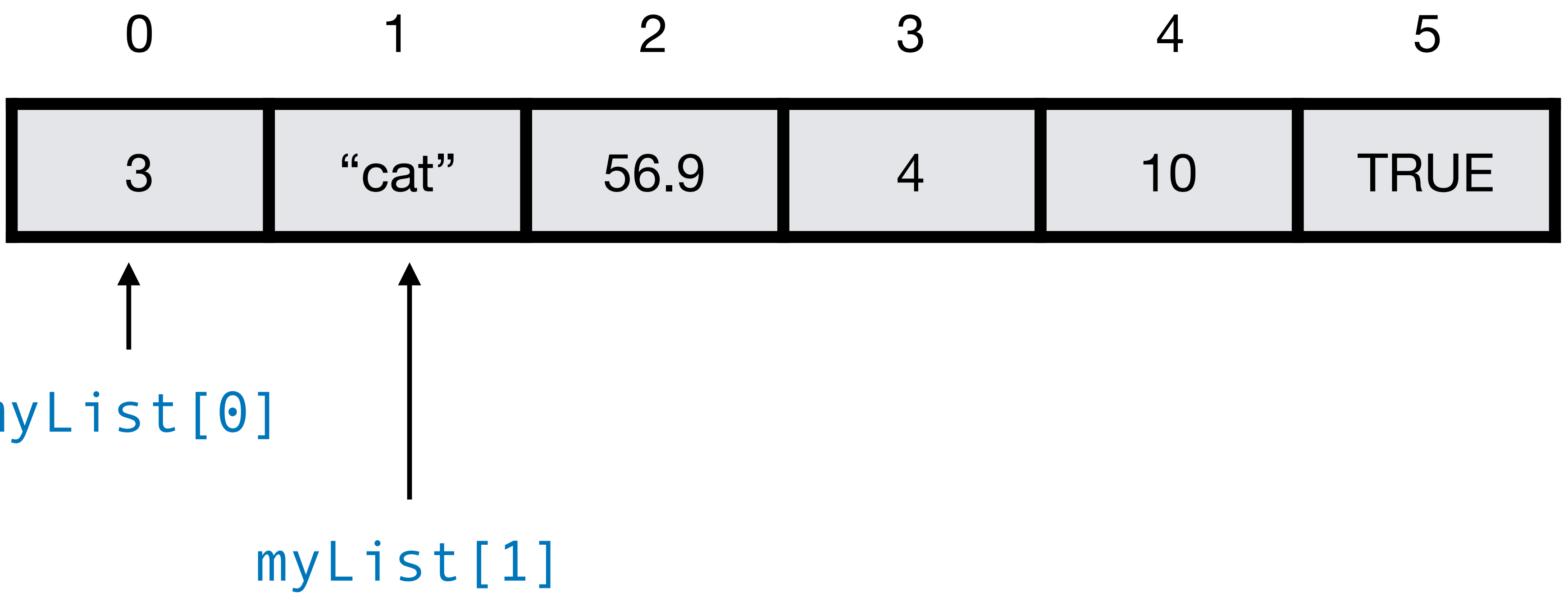
We use only one variable name to refer to the whole list. For example:

```
myList = [3, “cat”, 56.9, 4, 10, True]
```

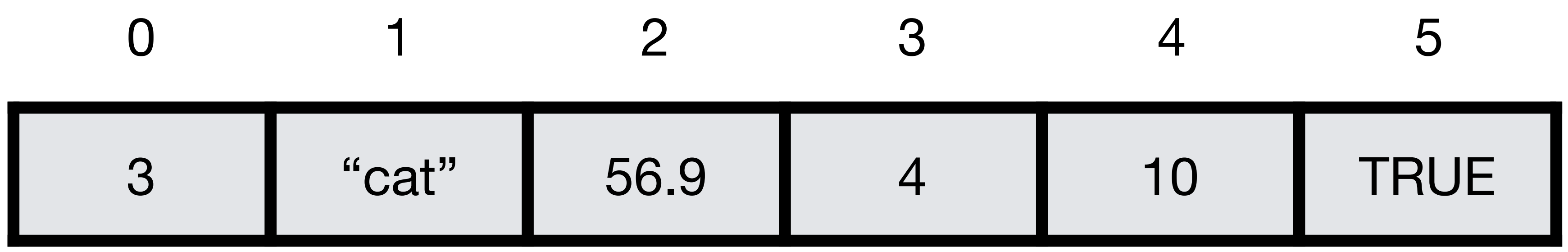
To access a specific element in the list, we use the following syntax:

```
listName[index]
```

```
>>> myList[0]
3
>>> myList[1]
cat
```



Practice with list indexing



What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[1]
```

Practice with list indexing

0	1	2	3	4	5
3	“cat”	56.9	4	10	TRUE

What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[4]
```

Practice with list indexing

0	1	2	3	4	5
3	“cat”	56.9	4	10	TRUE

What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[6]
```

Practice with list indexing

0	1	2	3	4	5
3	“cat”	56.9	4	10	TRUE

What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[-1]
```

Practice with list indexing

0	1	2	3	4	5
3	“cat”	56.9	4	10	TRUE

What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[-2]
```

Practice with list indexing

0	1	2	3	4	5
3	“cat”	56.9	4	10	TRUE

How would you get the third element?

Practice with list indexing

0	1	2	3	4	5
3	“cat”	56.9	4	10	TRUE

What will this code print?

```
myList = [3, “cat”, 56.9, 4, 9, 10, True]
myList[0] = “dog”
print(myList)
```

For-loops and Lists

- Last time we used `range()` to create `for` loops.
- In Python3 the type `range` is not the same as the type `list` (they are the same in Python2).
- That being said, we can use both to create `for` loops!

```
for i in [1, 20, 3, 19, 6]:
  print(i)
```

Output:

```
1
20
3
19
6
```

Practice with For-loops and Lists

What will the following code print?

```
for i in [“cat”, “dog”, “mouse”, “human”]:
  print(“I am a”, i)
```

Practice with For-loops and Lists

What will the following code print?

```
myStuff = [“cat”, 2, True, 99.5]
for i in myStuff:
  print(i)
```

Creating Lists

We now know how to:

- create an empty list

```
myList = []
```

- create a list of elements

```
myList = [2, 7, 8]
```

To create a list of numbers we can use the range function, but we must remember to specify it as a `list` or else will be caste as a `range` type.

```
myList = list(range(5, 50, 10))
```

recall that the format for `range()` is:
`range([start],stop[, step])`

Adding to a List

After creating a list, you can add additional elements to the **end** by using the `.append()` function

Syntax:

```
list.append(newElement)
```

Example:

```
>>>> myList = [2, 4, 6, 8]
>>>> myList.append(10)
print(myList)
[2, 4, 6, 8, 10]
```

Important to note:
Most of the functions we've seen so far do not modify variables directly -- they simply "return" a value. (e.g. `line.rstrip('\n')` does nothing to the original string -- it just returns a modified version. You have to say `line = line.rstrip('\n')` to actually change `line`.)
`.append()` is different. When you say `mylist.append()`, you are directly modifying `mylist`. We'll see several examples of this type of function today.

Removing from a list

After creating a list, you can remove elements from it using the `.pop()` function

Syntax:

```
list.pop(index)
list.pop()
```

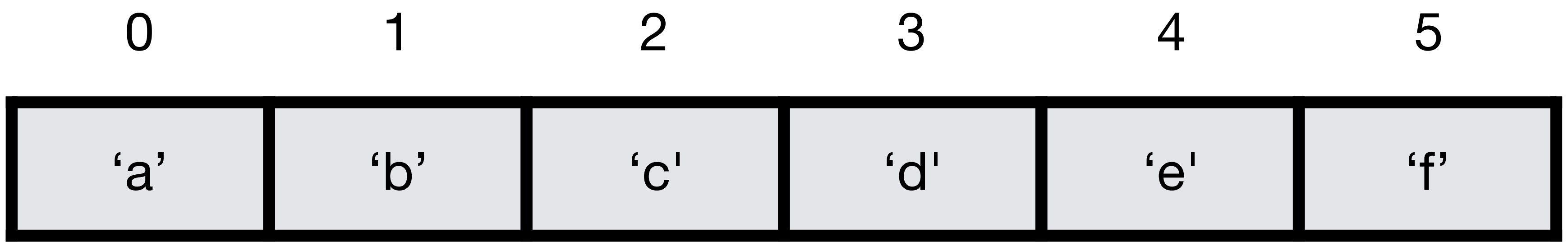
This in-place function removes the element at the specified index, or if no index is given, removes the last item. It also returns the removed item.

Example:

```
>>>> myList = [22, 44, 66, 88]
>>>> myList.pop(2)
print(myList)
[22, 44, 88]
```

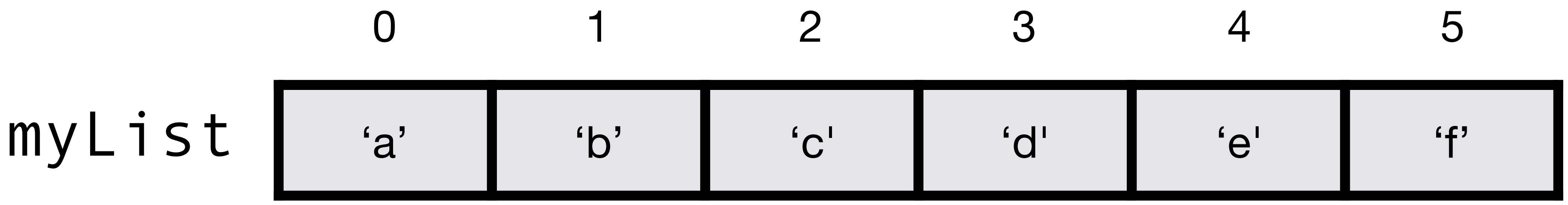
Elements that come after will be moved up one index, so that there are no empty spaces in the list.

Practice with lists



How do I add a ‘g’ to the end?

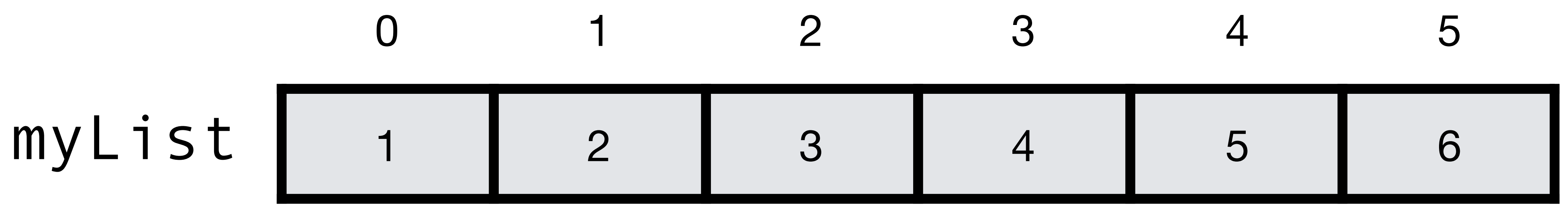
Practice with lists



What will this code print?

```
myList.pop(4)
print(myList)
```

Practice with lists



What will this code print?

```
myList.pop()
print(myList)
```

Practice with lists

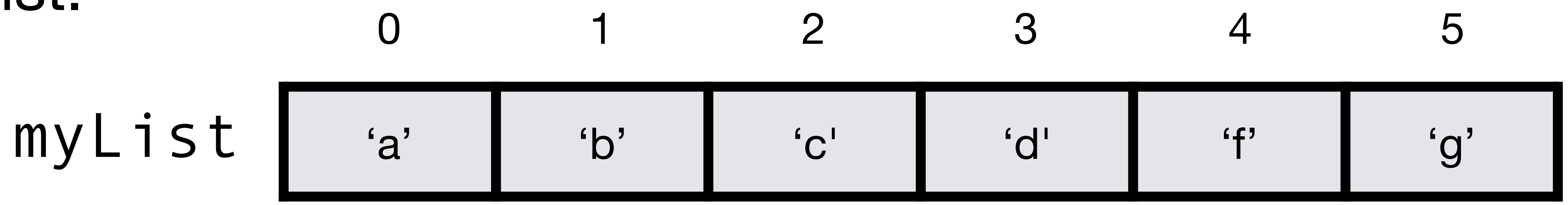
	0	1	2	3	4	5
`myList`	“24”	36	1	8	“cat”	True

What will this code print?

```
item = myList.pop()
print(item)
```

List slicing

Sometimes you may want to extract a certain subset of a list.



The syntax:
`list[begin:end]`
returns from index `begin` to `end-1`

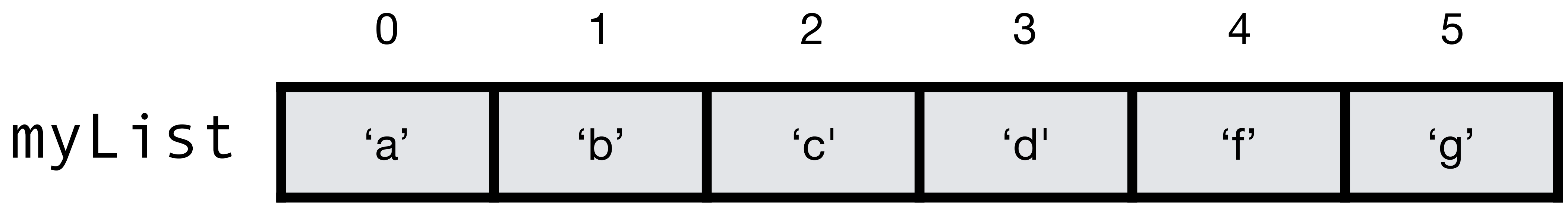
```
>>> myList = [‘a’,’b’,’c’,’d’,’f’,’g’]

>>> myList[2:]
[’c’,’d’,’f’,’g’]
>>> myList[:4]
[‘a’,’b’,’c’,’d’]

>>> myList[2:4]
[‘c’,’d’]
```

List slicing

You can also slice from the end of the list using negative indices



```
>>> myList = [‘a’,’b’,’c’,’d’,’f’,’g’]

>>> myList[-2:]
[’f’,’g’]
>>> myList[:-4]
[‘a’,’b’]

>>> myList[-4:-2]
[‘c’,’d’]
```

Side note: indexing strings like lists

Strings are NOT lists. But we can index into strings like we do lists:

```
>>> name = “Sammy”

>>> name[0]
’S’

>>> name[-1]
‘y’

>>> name[1:4]
‘amm’
```

Side note: indexing strings like lists

However, that being said, strings are not immutable (cannot be changed), so none of these operations are allowed:

```
>>> name = “Sammy”

>>> name[0]=“T”

TypeError: 'str' object does not support item
assignment

>>> name.append(“s”)

AttributeError: 'str' object has no attribute
'append'
```

Useful list functions

Lists come with several other helpful functions:

- `.sort()` - sorts **in place** (overwrites the list). Can sort both strings and numerical data.
- `.reverse()` - reverses order of items, **in place**
- `.index(element)` - returns index of the first occurrence of the specified element
- `.remove(element)` - Removes the first occurrence of the specified element. Elements that come after will shift down one index.
- `.insert(value, index)` - insert the value at the specified index. Elements that come after that index will shift up one index.
- `.count(element)` - returns the number of times the specified element occurs in the list

Functions that work on lists

There are also several built-in Python functions that work on lists:

- `len(list)` - returns the total number of elements in the list
- `max(list)` - returns the element in the list with the largest value
- `min(list)` - returns the element in the list with the smallest value
- `sum(list)` - returns the sum of the elements of the list

3. File parsing with .split()

The Situations

- You have a file with multiple columns separated by either tabs, commas, etc
- You want to extract certain columns of data to analyze
- How can you do this in Python?

`.split()`

- This functions splits a string into a list based on a delimiter.
- The delimiter can be anything you want, but usually it'll be a tab, space, or comma.
- This effectively lets you chop up a file into columns!

`.split()`

Purpose:
Splits a string every time it encounters the specified delimiter. If no delimiter is given, splits on whitespace (spaces, tabs, and newlines). The delimiter is not included in the output. If *maxsplit* is given, splits no more than *maxsplit* times. Returns a list.
Syntax:

```
  result = string.split([delimiter[,maxsplit]])
```

Example:

```
>>> sentence = "Hello, how are you today?"
>>> sentence.split()
['Hello,', 'how', 'are', 'you', 'today?']
```

Notice that the spaces are removed!

More examples

```
>>> sentence = "Hello, how are you today?"
```

```
>>> sentence.split(',')
['Hello', ' how are you today?']
```

Notice that now the comma is removed, but spaces are not!

```
>>> sentence.split(None, 2)
['Hello,', 'how', 'are you today?']
```

maxsplit must always be the second parameter. So if we don't want to specify a delimiter, we can put `None` instead as a placeholder

Why is `.split()` important?

This is perhaps the single most useful tool for parsing a text file (for what I do, anyway)

Let’s take a look at a real-life example.

A more realistic example: parsing a data file

A data file organized in rows and columns (data "table") can be easily parsed using a combination of a `for` loop and `.split()`.

Input Data File:

```
knownGene GeneName InitCodon DisttoCDS FramevsCDS InitContext CDSLength PeakStart PeakWidth #HReads PeakScore Codon Product

uc007zzs.1 Cbr3 36 -23 -1 GCCACGG 22 35 3 379 4.75 nearcog uorf

uc009akk.1 Rac1 196 0 0 CAGATGC 192 195 3 3371 4.70 aug canonical

uc009eyb.1 Saps1 204 -91 1 GCCACGG 23 203 3 560 4.68 nearcog uorf

uc008wzq.1 Ppp1cb 96 0 0 AAGATGG 327 94 4 3218 4.56 aug canonical

uc007hnl.1 Pa2g4 38 -23 0 AGCCTGT 14 37 4 6236 4.54 nearcog uorf

uc007hnl.1 Pa2g4 40 -22 -1 CCTGTGG 17 37 4 6236 4.54 nearcog uorf

uc008tvu.1 Leprot 27 0 0 GACATGG 131 26 3 830 4.53 aug canonical

uc008vlv.1 Capzb 95 0 0 ACCATGA 277 94 3 3024 4.51 aug canonical

uc007xgk.1 Ncaph2 63 -2 -1 GACATGG 38 62 3 983 4.48 aug uorf-overlap
```

A more realistic example: parsing a data file

Let's say I just want to extract the 6th column of each row (in this case, the initiation context for each start site).

Code:

```
inFile = "init_sites.txt"
input = open(inFile, 'r')
input.readline() #skip header
for line in input:
      line = line.rstrip('\n')
      data = line.split() #splits line on tabs
      print(data[5]) #6th column = index 5
input.close()
```

A more realistic example: parsing a data file

Let's say I just want to extract the 6th column of each row (in this case, the initiation context for each start site).

Code:

```
inFile = "init_sites.txt"
input = open(inFile, 'r')
input.readline() #skip header
for line in input:
      line = line.rstrip('\n')
      data = line.split() #splits line on tabs
      print(data[5]) #6th column = index 5
input.close()
```

Output:

```
GCCACGG
CAGATGC
GCCACGG
AAGATGG
AGCCTGT
CCTGTGG
GACATGG
ACCATGA
GACATGG
```

Appendix

Nested lists

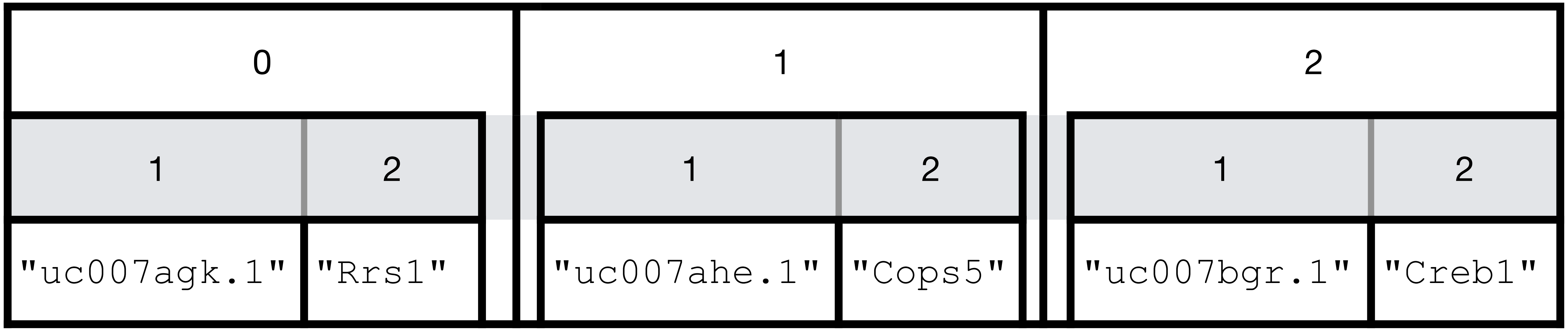
List comprehensions

Examples of `.insert()` and `.remove()`

Nested lists

A list can hold pretty much anything, including other lists:

```
>>> geneList = [["uc007agk.1", "Rrs1"], ["uc007ahe.1",
"Cops5"], ["uc007bgr.1", "Creb1"]]
```



```
>>> geneList[1]
['uc007ahe.1', 'Cops5']
>>> geneList[1][0]
'uc007ahe.1'
```

You can access individual items in a list of lists using double indexing:
`list[index][subindex]`

List comprehensions (advanced)

A list comprehension is just a quick, concise way of performing operations on the elements of a list. Returns a new list with the modified elements.

Syntax:

```
newList = [expression for item in list if condition]
```

Example:

```
>>> myList = [1, 2, 3, 4, 5]
>>> newList = [i * 2 for i in myList]
>>> newList
[2, 4, 6, 8, 10]
>>> newList = [i * 2 for i in myList if i > 3]
>>> newList
[8, 10]
```

List comprehensions (advanced)

Almost any function can be used as the *expression* part:

```
>>> myList = ["Joe", "Sally", "George", "Mike"]
>>> [len(i) for i in myList]
[3, 5, 6, 4]
>>>
>>> [i.upper() for i in myList]
['JOE', 'SALLY', 'GEORGE', 'MIKE']
>>>
>>> [(i == "George") for i in myList]
[False, False, True, False]
>>>
>>> [print(i) for i in myList]
File "<stdin>", line 1
[print(i) for i in myList]
^
SyntaxError: invalid syntax
```

Inserting into a list: `.insert()`

Purpose:

Insert new element at specified index. All elements after will be pushed back one index.

Syntax:

`list.insert(`*`index`*`, `*`newElement`*`)`

Example:

```
>>> myList = [2, 4, 6, 8]
>>> myList.insert(1, "hi!")
>>> print myList
[2, 'hi!', 4, 6, 8]
```

Practice with adding to lists

0	1	2	3	4	5
‘a’	‘b'	‘c’	‘d’	‘f’	‘g'

How do I insert an ‘e' between the ‘d’ and ‘f’?

Practice with adding to lists

0	1	2	3	4	5
‘a’	‘b'	‘c’	‘d’	‘f’	‘g'

How do I insert an ‘e' between the ‘d’ and ‘f’?

Answer:

```
myList.insert(4, “e”)
```

Remove element from a list: `.remove()`

Purpose:

Removes the first occurrence of the specified element.
Elements that come after will be moved up one index.

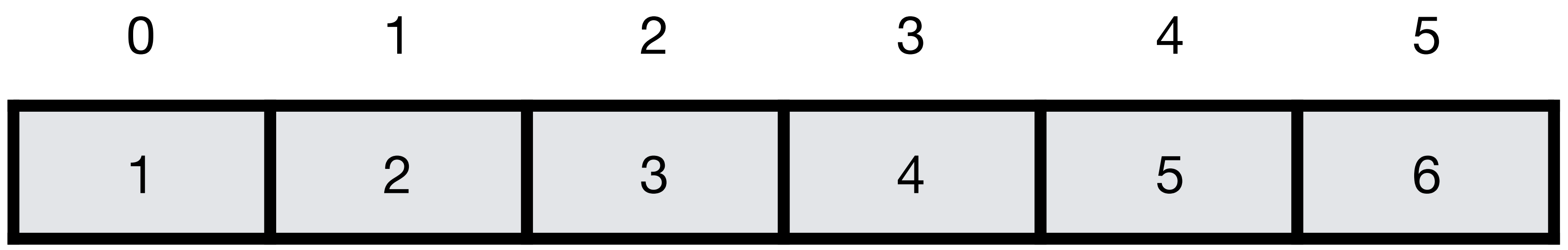
Syntax:

`list.remove(`*`element`*`)`

Example:

```
>>> myList = [22, 44, 66, 88]
>>> myList.remove(44)
>>> print myList
[22,66,88]
```

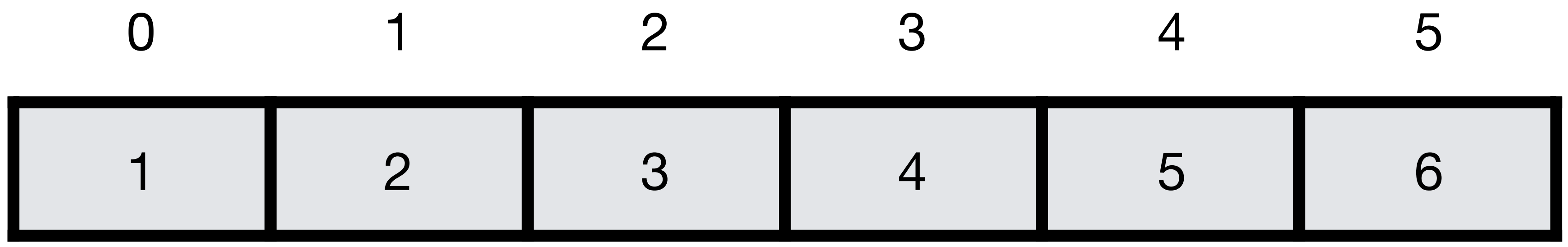
Practice with removing from lists



What will this code print?

```
myList.remove(4)
print myList
```

Practice with removing from lists



What will this code print?

```
myList.remove(4)
print myList
```

Answer

```
[1,2,3,5,6]
```