Scientific Computing 272

Section 4: Lists in Python

Last updated: 23 April 2019

UNIVERSITEIT STELLENBOSCH UNIVERSITY

Willem Bester

Section Outline

- Indices
- Modifying Lists
- List Functions
- Processing Items
- Slicing
- Aliasing
- Methods
- Nested Lists
- Sequences

Data Collections

- Up to now, we have put a single value into a single variable
- But what if we want to work with collections of data?

Example (Data collection)

The number of whales sighted near a research station, counted each day for a two-week period, is given in the next table.

Day:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Whales:	5	4	7	3	2	3	2	6	4	2	1	7	1	3

- Without a collection data type, we need fourteen variables to store these values
- This is still manageable, but what if have data for a year, or a decade, or a century?

Lists

- Solution: Use a **list**
- Put the values, separated by commas, inside (square) brackets

Example (List)

```
>>> [5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
[5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
```

- `list` is a Python data type
- Therefore, we can have objects of type `list`
- A list can be assigned to a variable
- A list object owns methods
- There are standard functions that operate on lists

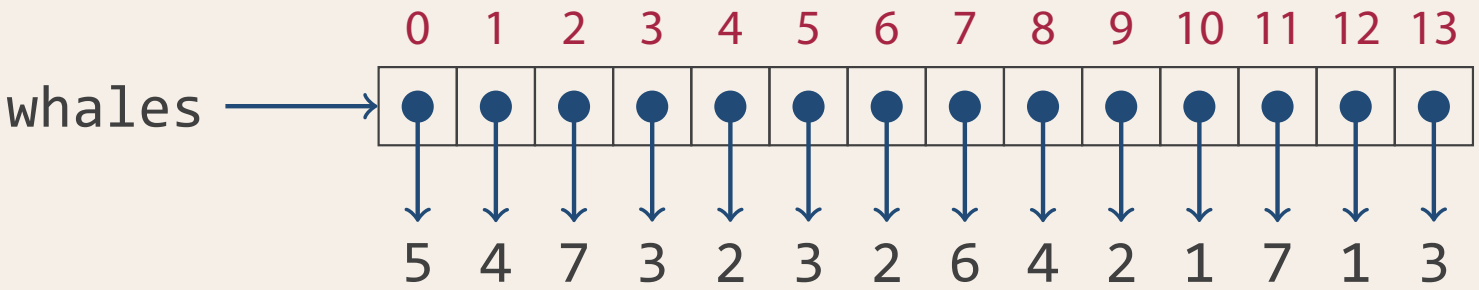
List Assignment

Example (Assign a list to a variable)

```
>>> whales = [5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
>>> whales
[5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
```

- The variable `whales` contains a **reference** to the list
- Each of the indices in the list contains a reference to a number

List Memory Model



List Indices

- Think of a list as a vector: Just as we can index over the vector components, we index over the elements in a list
- The first index is 0: Think of this as saying, "We are *x* positions from the front." Or: "There are *x* elements before this one."
- To refer to a particular item, put the index in brackets after a reference to the list, such as the name of a variable

Example (List indexing)

```
>>> whales = [5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
>>> whales[0]
5
>>> whales[11]
7
>>> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29][5]
13
```

Legal Indices

- For a list of length n, a legal index i is an integer in the set $\{0 \leq i < n\}$
- Trying to use an out-of-range index is an error

Indexing out of range

```
>>> whales = [5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
>>> whales[23]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Indexing Backwards

- Python also lets us index backwards from the end of a list
- Since 0 = −0, index −1 refers to the last item, and so on
- We can also assign the values in a list to other variables

Example (Indexing backwards)

```
>>> whales = [5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
>>> whales[-1]
3
>>> whales[-14]
5
>>> third = whales[2]
>>> print('On the third day', third, 'whales were seen.')
On the third day 7 whales were seen.
```

The Empty List

- In maths and computer science, we note identity elements
- For example, 0 for arithmetic, and the empty string for strings
- There is also an empty list, written `[ ]`, with no elements

An empty list has no legal indices

```
>>> whales = []
>>> whales[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> whales[-1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Legal Indices for the Empty List

- Trying to index into an empty list is always an error
- Legal indices i for a list of length n are in the set $\{i \in \mathbb{Z} \mid -n \leq i < n\}$
- For an empty list, $n = 0$
- So, a legal index i must be in $\{0 \leq i < 0\}$
- This set is empty
- So, there are no legal indices into an empty list

Lists Are Heterogeneous

- Lists can contain any type of data
- We can also “mix” different data types in one list
- Therefore, lists are called **heterogeneous**

Example (List heterogeneity)

```
>>> kr = ['Krypton', 'Kr', -157.2, -153.4]
>>> print(kr[0], 'boiling point is', kr[3], 'centigrade')
Krypton boiling point is -153.4 centigrade
```

- Using a list to aggregate data is not a good idea
- (We shall see better ways later)
- Besides, many list functions assume all items have the same type, and these functions fail if the items do not

Modifying Lists

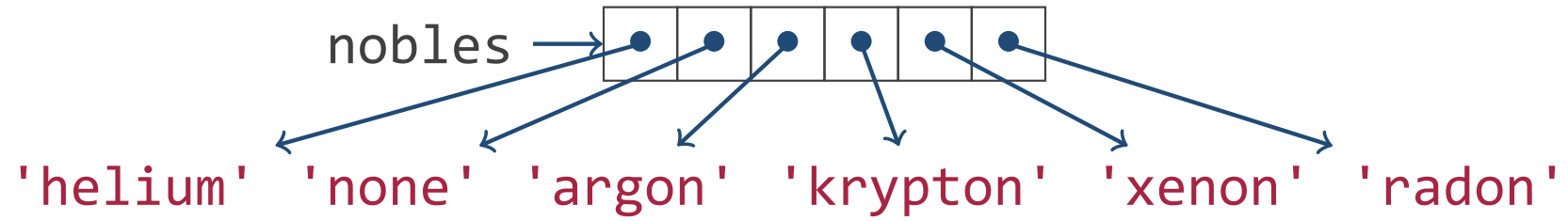
- Lists are **mutable**
- We can modify a list after it has been declared

Example (List mutation)

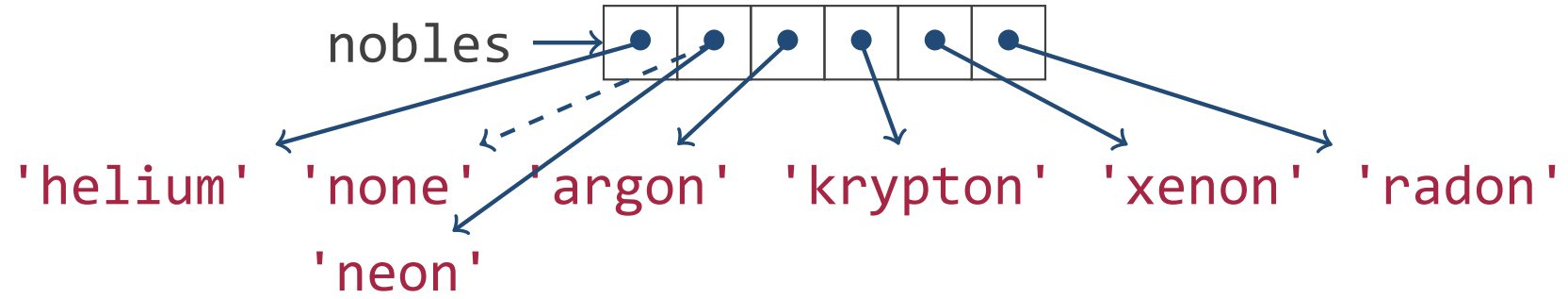
```
>>> nobles = ['helium', 'none', 'argon', 'krypton',
... 'xenon', 'radon']
>>> nobles
['helium', 'none', 'argon', 'krypton', 'xenon', 'radon']
>>> nobles[1] = 'neon'
>>> nobles
['helium', 'neon', 'argon', 'krypton', 'xenon', 'radon']
```

List Mutation

Before mutation



After mutation



Strings Are Immutable

- The expression `L[i]`, where `L` is a list and `i` an index, behaves just like a normal variable
- On the right: "Get the value of the item at `i` in `L`"
- On the left: "Figure out where item `i` is in `L` so that we can overwrite it"
- Compare this to strings, where we cannot change a letter after the string has been created
- For example, `upper()` actually creates a new string

Example (Strings are immutable)

```
>>> name = 'Mendeleev'
>>> capitalized = name.upper()
>>> print(name, capitalized)
Mendeleev MENDELEEV
```

Built-In List Functions

Table: Built-in list functions

Function	Description
`len(L)`	Returns the number of items in list `L`
`max(L)`	Returns the maximum value in list `L`
`min(L)`	Returns the minimum value in list `L`
`sum(L)`	Returns the sum of the values in list `L`

- `max` and `min` use the **natural order** of the list elements, which have to be mutually comparable
- Note that we have seen some of these functions before
- `len`, for example, has previously been applied to strings

List Functions and Range Checking

Example (Half-lives of Plutonium)

```
>>> half_lives = [87.74, 24110.0, 6537.0, 14.4, 376000.0]
>>> len(half_lives)
5
>>> max(half_lives)
376000.0
>>> min(half_lives)
14.4
>>> sum(half_lives)
406749.14000000001
>>> i = 2
>>> 0 <= i < len(half_lives)
True
```

Note how we check whether an index is in range.

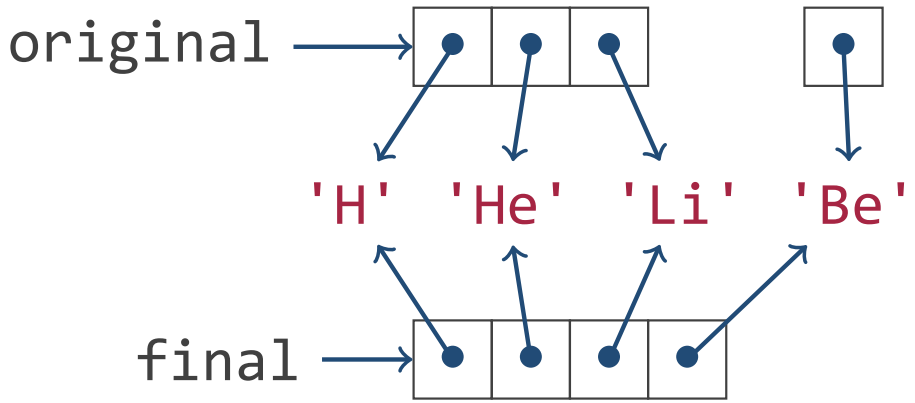
List Operations

Inappropriate list concatenation

```
>>> ['H', 'He', 'Li'] + 'Be'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
```

- Python complains if we try to combine a list with other types in inappropriate ways
- It is not possible to append a string to a list with the + operator
- But the + operator is overloaded for lists, so we can concatenate lists just like we can concatenate strings…

Memory Model for List Concatenation



Example (List concatenation)

```
>>> original = ['H', 'He', 'Li']
>>> final = original + ['Be']
>>> final
['H', 'He', 'Li', 'Be']
```

List Functions and Methods

Example

```
>>> 1 + 2 + 3
6
>>> sum([1, 2, 3])
6
>>> sum(['a', 'b', 'c'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> metals = 'Fi Ni'.split()
>>> metals * 3
['Fi', 'Ni', 'Fi', 'Ni', 'Fi', 'Ni']
```

- `sum` requires its argument list to contain numbers
- Note how we use the `split` method of a string to turn the string `'Fe Ni'` into a two-element list `['Fe', 'Ni']`

Iteration and `for` Loops

- Python has lists so that we don't need to create *n* variables to store *n* values
- A `for` loop lets us **iterate** over a list, without having to write one statement per element
- We call it a loop, because it **repeats** a block of statements

General Form of a `for` Loop

```
for ⟨variable⟩ in ⟨iterable⟩:
    ⟨block⟩
```

- ⟨**iterable**⟩ is an **iterable object** like a list
- ⟨**variable**⟩ is a variable that takes on each of the values in the iterable object in turn
- ⟨**block**⟩ is a block of statements

What Happens in a `for` Loop

- Python executes the loop block once for each value in the iterable object
- Each pass through the block is called an **iteration**
- At the start of each iteration, Python assigns the next value in the iterable object to the variable
- So, we can do something with each value in turn
- Remember that the block statements must be indented
- In English, we would say: "For each element in the iterable object, perform the block of statements"

Looping with `for`

Example (`for` loop)

```
>>> velocities = [0, 9.81, 19.62]
>>> for v in velocities:
...     print('Metric', v, 'm/s;',
...           'Imperial', v * 3.28, 'ft/s')
...
Metric 0 m/s; Imperial 0.0 ft/s
Metric 9.81 m/s; Imperial 32.1768 ft/s
Metric 19.62 m/s; Imperial 64.3536 ft/s
>>> print('v is now {}'.format(v))
v is now 19.62
```

- Note that `v` functions like a normal variable
- So, we can perform arithmetic, etc.
- After the `for` loop has completed, the variable refers to the value it was assigned during the last execution of the loop

Nested Loops

Example (Nested Loops)

```
>>> outer = ['Li', 'Na', 'K']
>>> inner = ['F', 'Cl', 'Br']
>>> for metal in outer:
...     for gas in inner:
...         print(metal + gas, end=' ')
...     print()
...
LiF LiCl LiBr
NaF NaCl NaBr
KF KCl KBr
>>> print('metal = {}, gas = {}'.format(metal, gas))
metal = K, gas = Br
```

The Basic Counting Principle: Number of Loop Iterations

If the outer loop runs n_{outer} times and the inner loop runs n_{inner} times for each of them, the inner loop executes $n_{\text{outer}} \times n_{\text{inner}}$ times.

Ranges

Use the `range` function to generate integer values in a specified range; it works only for `int`s. Of course, these statements don't necessarily have to do anything with either the iterable object or the values…

Example (Repetition)

```
>>> for i in range(3):        # 0 <= i < 3
...     print(i)
...
0
1
2
>>> for i in range(3):        # 0 <= i < 3
...     print('Hi there!')
...
Hi there!
Hi there!
Hi there!
```

Ranges

The `range` function can be called in three ways:

- `range(n)` for the values $0 \leq i < n$ with step size 1
- `range(m, n)` for the values $m \leq i < n$ with step size 1
- `range(m, n, s)` for the values $0 \leq i < n$ with step size s

Example (Repetition)

```
>>> for i in range(-3, 0):
...     print(i)
...
-3
-2
-1
>>> for i in range(4, 13, 3):
...     print(i)
...
4
7
10
```

Nested Loops

Example (`multiplication_table.py`)

```
def print_table(n):
    """Print the multiplication table for numbers 1 to n."""

    # print the headers row
    for i in range(1, n + 1):
        print('\t{}'.format(i))
    print()  # end the header row

    # print the column number and the contents of the table
    for i in range(1, n + 1):
        print(i, end=' ')
        for j in range(1, n + 1):
            print('\t{}'.format(i * j), end=' ')
```

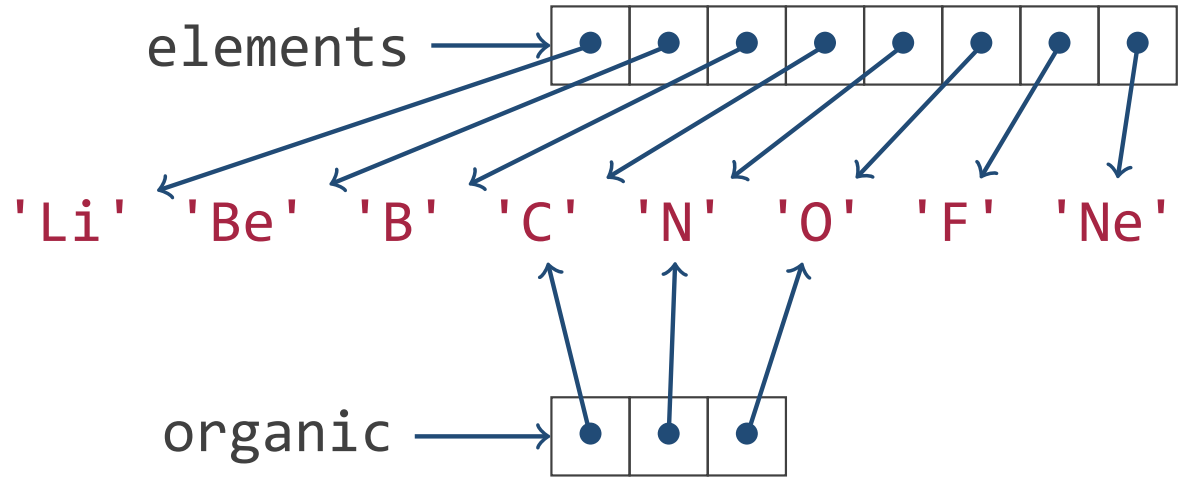
Slicing

- We can create a new list from an existing list by taking a **slice**
- For `list`, `list[i:j]` is the slice of the original list, from index `i` (inclusive) to index `j` (exclusive)

Example (List slicing)

```
>>> elements = ['Li', 'Be', 'B', 'C',
                'N', 'O', 'F', 'Ne']
>>> elements[1:6]
['Be', 'B', 'C', 'N', 'O']
>>> organic = elements[3:6]
>>> organic
['C', 'N', 'O']
```

Slicing Memory Model



- Slicing does not modify lists
- That means: Slicing leaves the original list intact
- And slicing returns a new list

List Slicing and Copying

- We may omit the first or last indices if we want to slice from the beginning or end, respectively
- We may omit both indices to obtain a copy of the original

Example (List slicing and copying)

```
>>> elements = ['Li', 'Be', 'B', 'C', 'N', 'O', 'F', 'Ne']
>>> elements[3:]
['C', 'N', 'O', 'F', 'Ne']
>>> elements[:5]
['Li', 'Be', 'B', 'C', 'N']
>>> copy = elements[:]
>>> elements.append('Na')
>>> elements
['Li', 'Be', 'B', 'C', 'N', 'O', 'F', 'Ne', 'Na']
>>> copy
['Li', 'Be', 'B', 'C', 'N', 'O', 'F', 'Ne']
```

Aliasing

- An **alias** is an alternative for something
- In Python, two variables are said be **aliased** if they refer to the same object, that is, contain the same reference
- If two variables contain a reference to the same list, modifying the list using one variable will be "seen" by the other

Example (Aliasing)

```
>>> elements = ['Li', 'Be', 'B', 'C', 'N', 'O', 'F', 'N']
>>> elements_copy = elements
>>> elements[7] = 'Ne'
>>> elements
['Li', 'Be', 'B', 'C', 'N', 'O', 'F', 'Ne']
>>> elements_copy
['Li', 'Be', 'B', 'C', 'N', 'O', 'F', 'Ne']
```

Aliasing in Function Calls

Example (List aliasing in functions)

```
>>> def sort_and_reverse(L):
...     '''Return list L sorted and reversed.'''
...     L.sort()
...     L.reverse()
...     return L
...
>>> elements = ['Li', 'Be', 'B', 'C', 'N', 'O', 'F', 'N']
>>> e2 = sort_and_reverse(elements)
>>> e2
['O', 'N', 'N', 'Li', 'F', 'C', 'Be', 'B']
>>> elements
['O', 'N', 'N', 'Li', 'F', 'C', 'Be', 'B']
>>> e2[3] = 'Oops'
>>> e2
['O', 'N', 'N', 'Oops', 'F', 'C', 'Be', 'B']
>>> elements
['O', 'N', 'N', 'Oops', 'F', 'C', 'Be', 'B']
```

List Methods

Method	Description
`L.append(v)`	Appends value `v` to list `L`, i.e. `L[len(L):] = [v]`
`L.extend(M)`	Appends all items in the iterable object `M` to the end of list `L`, i.e. `L[len(L):] = M`
`L.index(v)`	Returns the index of the first item whose value is `v`
`L.insert(i, v)`	Inserts value `v` at index `i` in list `L`, shifting following items to make room
`L.pop()`	Removes and returns the last element of `L`, which must be nonempty; `L.pop(i)` removes and returns the item at index `i`
`L.remove(v)`	Removes the first occurrence of value `v` from list `L`
`L.reverse()`	Reverses the order of the values in list `L`
`L.sort()`	Sorts the values in list `L` in ascending order

For sorting, Python uses **Timsort**, which is **adaptive**, meaning sorting is faster if list elements are sorted or almost sorted, and **stable**, meaning equal elements appear in the same order in the sorted list as they did in the unsorted list.

List Methods

Example (List methods)

```
>>> colours = 'red orange green black blue'.split()
>>> colours
['red', 'orange', 'green', 'black', 'blue']
>>> colours.remove('black')
>>> colours
['red', 'orange', 'green', 'blue']
>>> colours.insert(2, 'yellow')
>>> colours
['red', 'orange', 'yellow', 'green', 'blue']
>>> colours.append('purple')
>>> colours
['red', 'orange', 'yellow', 'green', 'blue', 'purple']
```

List Methods

- Remember that a list method modifies a list—it does not return a new list
- List methods may return the special value None; Python does not display anything when asked to evaluate None

Example (The special value None)

```
>>> x = None
>>> x
>>> print(x)
None
>>> colours = 'red yellow blue green'.split()
>>> colours
['red', 'yellow', 'blue', 'green']
>>> sorted_colours = colours.sort()
>>> sorted_colours
>>> colours
['blue', 'green', 'red', 'yellow']
```

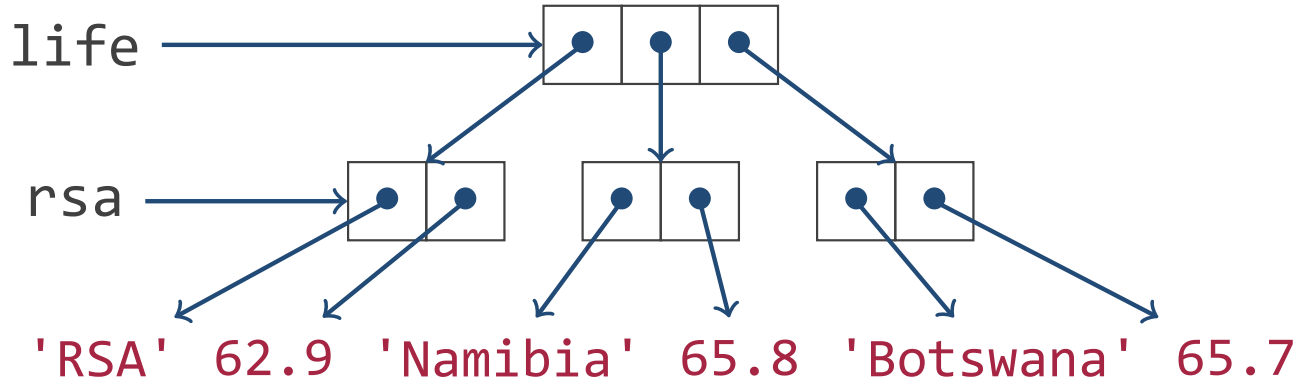
Nested Lists

- Since lists are heterogeneous, they can contain other lists
- We can also assign sublists to variables

Example (Nested lists)

```
>>> life = [['South Africa', 62.9],
...         ['Namibia', 65.8],
...         ['Botswana', 65.7]]
>>> life[0]
['South Africa', 62.9]
>>> print('Life expectancy in', life[1][0], 'is', life[1][1])
Life expectancy in Namibia is 65.8
>>> botswana = life[2]
>>> botswana[0]
'Botswana'
>>> botswana[1]
65.7
```

Aliasing in Nested Lists



Assigning a sublist to a variable creates an alias, so changes show up in the main list.

Example (Aliasing in nested lists)

```
>>> life = [['ZA', 62.9], ['NA', 65.8], ['BW', 65.7]]
>>> rsa = life[0]
>>> rsa[1] = 63
>>> life
[['RSA', 63], ['NA', 65.8], ['BW', 65.7]]
```

Other Sequences: Strings

- Formally, a string is an **immutable sequence** of characters
- Being a sequence, a string can be indexed and sliced

Example (Strings as sequences)

```
>>> rock = 'anthracite'
>>> rock[9]
'e'
>>> rock[0:3]
'ant'
>>> for character in rock[:5]:
...     print(character, end=' ')
...
a n t h r
```

Other sequences: Tuples

- A **tuple** is a general sequence like a list, but unlike lists, they are immutable
- Tuples are written with parentheses instead of brackets
- The empty tuple is written `()`
- To avoid ambiguity, a tuple with one element `x` is written `(x,)`

Example (Tuples)

```
>>> bases = ('A', 'C', 'G', 'T')
>>> for base in bases:
...     print(base, end=' ')
...
A C G T
```

Immutability of Tuples

Although tuples cannot be changed once created, the objects they refer to can still be changed—if these referenced objects are, themselves, mutable.

Tuples are immutable

```
>>> life = [['ZA', 62.9], ['NA', 65.8], ['BW', 65.7]]
>>> life[0] = life[1]
>>> life = (['ZA', 62.9], ['NA', 65.8], ['BW', 65.7])
>>> life[0] = life[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> life[0][1] = 63
>>> life
(['ZA', 63], ['NA', 65.8], ['BW', 65.7])
```

The `sys` Module and Command-Line Arguments

- The `sys` module provides access to variables used and maintained by the Python interpreter
- It also contains functions that interact "strongly" with the interpreter
- `sys.argv` is the list of command arguments passed to a Python script
- This means we don't have to run Python interactively or use `input` to get input from a user
- Careful: All items in `sys.argv` are strings, so convert them to appropriate numeric types if necessary
- Also, `sys.argv[0]` **always** contains the name of the script, including the `.py` extension if present

The sys Module and Command-Line Arguments

Example (print_cmd.py)

```
import sys

if __name__ == '__main__':
    for i in range(len(sys.argv)):
        print(i, sys.argv[i], type(sys.argv[i]))
```

Example (Use of sys.argv)

```
whkbester@h00:~$ python print_cmd.py calculate math.sqrt 22.0
0 print_cmd.py <type 'str'>
1 calculate <type 'str'>
2 math.sqrt <type 'str'>
3 22.0 <type 'str'>
```