COMP 364: Computer Tools for Life Sciences

Python programming: Lists

Christopher J.F. Cameron and Carlos G. Oliver
Key course information

Quiz #1 is now available!
- Available on MyCourses (multiple choice questions)
- Quiz #1 closes at 11:59:59 pm on Monday, September 18th
- Multiple choice questions covering topics from the last two weeks
- Quizzes should be completed individually

Assignment #1 is now available!
- Available through the course website: http://cs.mcgill.ca/~cgonza11/COMP_364/
- Due: September 29th at 11:59:59 pm
- Please start early and see the TAs for help
Abstraction
Abstract data types (ADT) are logical descriptions of how to view data and operations allowed regardless of implementation

- Concerned only with what the data is representing
- Not with how it will eventually be constructed

ADTs allow us to separate specification from implementation

- Specification: what kind of thing are we working on? what operations can be performed?
- Implementation: how the thing and its operations are coded

ADTs make programs easier to understand and modify

- Which helps makes a program good
The implementing of an ADT is often referred to as a **data structure**

- Provides a physical view of the data using some collection of programming constructs and primitive data types
- Allows for an implementation-independent view of the data
  - Many different ways to implement an ADT

Implementation independence allows the programmer to switch the details of the implementation

- Without changing the way the user interacts with it
- The user can remain focused on the problem-solving process
Sequences

The most basic data structure in Python is the **sequence**

- Each element of a sequence is assigned a number - its position or index
- If the length of the sequence is $N$
  - The first index is zero
  - The second index is one
  - ...
  - The last index is $N - 1$

The two most common types of sequences are:

- Lists - dynamic in size and **mutable**
- Tuples - fixed size and **immutable**
Some objects in Python are **mutable**
- Meaning that they can be altered
- E.g., lists, dictionaries, sets

While others are **immutable**
- Cannot be changed but rather return new objects when attempting to update
- E.g., int, float, tuple, bool, string

When does mutability matter?
Example of a very inefficient use of memory:

```python
string = ""
# strings: ["""]
string += "How"
# strings: ["","How","How"]
string += " are"
# strings: ["","How","How","are","How are"]
string += " you?"
# strings: ["","How","How","are","How are","How are you?"]
```

Strings are **immutable**
- Concatenating two strings together actually creates a third
  - Which is the combination of the previous two
Lists vs. tuples

Lists

- When you don’t have a constant set of values
  - I.e., unknown length
  - Allows you to add/remove items from the sequence

Tuples

- Are faster than lists
- If you’re defining a constant set of values
  - All you’re ever going to do is iterate through it
  - Use a tuple instead of a list
- Code is safer if you ’write-protect’ data that does not need to be changed
Creating lists

To create a list:

1. `L = []` # empty list
2. `L = [expression, ...]`

Python also has a built-in **list** type to create lists:

1. `L = list()`
2. `L = list(sequence)`

The sequence can be any kind of sequence object, including tuples.

If you pass in another list, the **list** function makes a copy.
Creating lists #2

Python creates a new list every time the [ ] expression is executed

- Python never creates a new list if you assign a list to a variable

```python
1 A = B = []  # both variables point to the same list
2
3 A = []
4 B = A  # both variables point to the same list
5
6 A = []; B = []  # independent lists
```

11 / 23
Printing lists

To print a list in Python

```python
L = [0,1,2,3,4,5,6,7,8,9]
print(L) # prints ' [0,1,2,3,4,5,6,7,8,9]'
```

.join() allows us to format the print statement

- As long as all items are strings

```python
"".join(L) # prints '123456789'
",".join(L) # prints '1,2,3,4,5,6,7,8,9'
```
Accessing lists

`len(L)` will return the number of items in a list

1. \( L = [0,1,2,3,4,5,6,7,8,9] \)
2. \( n = \text{len}(L) \)
3. \( \text{print}(n) \) # prints '10'

\( L[i] \) returns the sequence item at index \( i \)

- The first item in a list has index 0
- The last item in a list has index \( n - 1 \)

1. `first_item = L[0]`
2. `last_item = L[9]`
3. `print(last_item is L[-1])` # prints 'True', why?

If you pass in a negative index, Python adds the length of the list to the index
**Accessing lists #2**

$L[i:j]$ will return a new list that contains all items between $i$ and $j$

```python
1  seq = L[0:3]
2  print(seq)  # prints ??
```

Lists also support **slicing**:

```python
1  seq = L[start:stop:step]
2  seq = L[::2]  # get every other item, starting with the first
3  seq = L[1::2]  # get every other item, starting with the second
```

**Note:** slicing is not inclusive of the stop index, $[start:stop)$

If an index is outside the list, Python raises an `IndexError` exception
Modifying lists

Lists allow for the assignment of individual items or slices

L[i] = obj  # obj can be an int, string, etc.
L[i:j] = sequence

Operations that modify the list will modify it in place

▶ if multiple variables point to the same list, all variables will be updated

L = [0,1,2,3,4,5,6,7,8,9]
M = L  # M points to L

# modify only L
index=3
L[index] = obj
print(M[index])  # prints value of 'obj'
Adding items to a list

**.append()** adds a single item to the end of the list

**.extend()** adds items from another list to the end of the list

**.insert()** inserts an item at a given index
  - Moves the remaining items to the right

1. L.append(item)
2. L.extend(sequence)
3. L.insert(index, item)
Deleting items from a list

**del** statement can be used to remove an item or slice of items

```
1    del L[i]
2    del L[i:j]
```

**.pop()** will remove an individual and return it

```
1    item = L.pop()  # last item
2    item = L.pop(0)  # first item
3    item = L.pop(index)
```

**del** statement and **.pop()** behave quite similar, except **.pop()** returns the removed item
Deleting items from a list #2

.. remove() removes the first instance of a matching item in a list

```python
1 L = [0,1,2,3,4,5,6,3,7,8,9]
2 L.remove(3)
3 print(L)  # prints '[0,1,2,4,5,6,3,7,8,9]'
```

If no matching item is found in the list, Python raises a ValueError exception
Reversing the order of a list

`.reverse()` allows you to quickly reverse the order of a list

```
L = [0,1,2,3,4,5,6,7,8,9]
L.reverse()
print(L)  # prints '[9,8,7,6,5,4,3,2,1,0]'
```

Reversing is fast

- Temporarily reversing a list can often speed things up
- Remove and insert many items at the end of the list

```
L.reverse()
# append/insert/pop/delete at far end
L.reverse()
```
Searching lists

`index()` returns the index of the first matching item in a list

```python
L = [1,2,3,4,5,5,6,7,8,9,10]
index = L.index(5)
print(index)  # prints '4'
```

`index()` performs a linear search, and stops at the first match

- If no matching item is found, Python raises a `ValueError` exception
.sort() sorts a list in place

```
1 L = [2,1,3,4,5,1,6]
2 L.sort()
3 print(L) # prints '[1,1,2,3,4,5,6]'
```

If you require a copy of the sorted list, use the `sorted()` function

```
1 L = [2,1,3,4,5,1,6]
2 sorted_L = sorted(L)
3 print(L) # prints '[2,1,3,4,5,1,6]'
4 print(sorted_L) # prints '[1,1,2,3,4,5,6]'
```
Other useful functions/methods

\textbf{min()} returns the smallest item in a list

\begin{verbatim}
L = [0,1,2,3,4,5,6,7,8,9]

print(min(L))  # prints '0'
\end{verbatim}

\textbf{max()} returns the largest items in a list

\begin{verbatim}
print(max(L))  # prints '9'
\end{verbatim}

In the future, we will learn how to apply these functions to non-integer sequences
Mutability example #2

A more efficient and pythonic way:

```python
L = [""]
L.append("How")    # L = ["", "How"]
L.append(" are")   # L = ["", "How", " are"]
L.append(" you?")  # L = ["", "How", " are", " you?"]
print(''.join(L))  # prints 'How are you?'
```

Takes advantage of a single mutable list object to gather strings

- Then allocates a single result string to store data
- Reduces the total number of objects allocated by almost half

pythonic: code follows proper syntax and conventions of the Python community

- Uses the language in the way it is intended to be used