Data Structures

Lists, Tuples, Sets, Dictionaries

Collections

- Programs work with simple values: integers, floats, booleans, strings
- Often, however, we need to work with collections of values (customers, books, inventory items, etc.)
- Python supplies four kinds of collections: lists, tuples, sets, and dictionaries
- Collections, like simple values, can be stored in variables
- There are two basic approaches to working with a collection:
  1. You can select specific individual elements from the collection and work with them individually
     - This often involves a loop, to select each element in turn
  2. You can work with the collection as a whole, producing a new collection
     - We will look at how to do this later in the course

Lists

- A *list* is a collection of zero or more elements, enclosed in brackets and separated by commas
  - All the elements must be of the same type, for example, all Strings
  - Example: `my_list = ["one", "two", "three"]`
- To access a single element in a list, give the list, then in brackets give the *index* of the desired element, ***starting from zero***
  - `my_list[0]` is `"one"`, `my_list[1]` is `"two"`, and `my_list[2]` is `"three"`
- You can also use ***negative*** indices
  - `my_list[-1]` is `"three"`, `my_list[-2]` is `"two"`, and `my_list[-3]` is `"one"`
- Lists are ***mutable***: You can change their contents
  - `my_list[-1] = "many"`
- It is an error to index `my_list` with an integer outside the range `-3` to `2`

List elements

- The elements of a list need not be simple values; they can be lists, tuples, sets, dictionaries, or any other kind of complex value
  - The elements must always be of the same type
- Example: `names = [["Mary", "Jane", "Sally", "Ann"], ["Tom", "Dick", "Harry"]]`
  - The elements of the above list (`names`) are themselves lists, and can be accessed as such
  - `names[0]` is `["Mary", "Jane", "Sally", "Ann"]`
  - `names[0][2]` is `"Sally"`

Functions and methods

- An ***object*** consists of some data (describing the ***state*** of the object) and some operations on that data (which may change the state of the object)
  - Lists, sets, tuples, and dictionaries are kinds of objects
- A ***function*** is an operation that is not part of an object
  - A function may take objects as arguments, or produce them as results, but it isn't part of those objects
  - **Syntax:** *function_name*`(`*arg*`, ..., `*arg*`)`
  - **Example:** `len(my_list)` returns the number of elements in `my_list`
- A ***method*** is an operation that "belongs to" an object
  - You must specify which object you are "talking to"
  - **Syntax:** *object*`.`*method_name*`(`*arg*`, ..., `*arg*`)`
  - **Example:** `my_list.sort()` sorts `my_list`
- **Mnemonic:** Methods Modify objects; functions don't.

List methods

- Several list methods are given in your textbook
  - You can add and remove items, search, sort, and reverse
- If you have a list `my_list` and you want to see what methods are applicable to it, enter
  `my_list.`
  in IDLE and wait a few seconds after typing the dot
  - You can select from the list that appears by hitting **Tab** (not Enter!)
  - On my computer, if I hit Enter, IDLE hangs temporarily, but recovers
- If you want to know what arguments to give to a method, or what it does, type the method name and the opening parenthesis...
  - **Example:** `my_list.append(`
  - ...and you will see what to do next; again, this is a feature of IDLE

An easy error to make

- Remember, Methods Modify objects (not always, but often)

-
  ```
  >>> my_list = ['one', 'two']
  >>> bigger_list = my_list.append('three')
  >>> print(bigger_list)
  None
  ```
- What happened?
- `append` ***modified*** the original list, but ***returned*** `None`

-
  ```
  >>> print(my_list)
  ['one', 'two', 'three']
  ```

Looping with indices

- The function `len(`*list*`)` returns the number of elements in *list*
- `range(`*start*`, `*end*`)` generates a series of integers starting with *start* and going up to, but not including, *end*
  - `range` is often used to generate a sequence of list indices
  - A range ***isn't*** a list, but it can be turned into a list by using it as an argument to the `list` function
- A *for loop* has the syntax

```
for variable in list_or_range:
        statements
```

- The combination of `for`, `len`, and `range` is often used to process every element of a list

  -
    ```
    for index in range(0, len(my_list)):
      print("Element", index, "is", my_list[index])
    ```

Looping without indices

- If you don't need to know the index of every list element that you process, it's better to use the form of **`for`** loop that accesses the list elements directly

  -
    ```
    for element in my_list:
     print(element, "is in the list")
    ```

Slices

- You can get a slice of a list using the syntax *list*`[`*from*`:`*upto*`]`; this is a **new** list containing the elements starting at *from* and going up to (but not including) *upto*

-
  ```
  >>> numbers = [0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
  ```

-
  ```
  >>> numbers[2:4]
  [2.0, 3.0]
  ```

-
  ```
  >>> numbers[:4]
  [0.0, 1.0, 2.0, 3.0]
  ```

-
  ```
  >>> numbers[2:]
  [2.0, 3.0, 4.0, 5.0]
  ```

-
  ```
  >>> numbers[3:6]
  [3.0, 4.0, 5.0]
  ```

-
  ```
  >>> numbers[2:][1]
  3.0
  ```

-
  ```
  >>> numbers
  [0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
  ```

Indexing into strings

- A string isn't a list, but it can be indexed like one

  -
    ```
    >>> "Computer"[0]
    'C'
    >>> "Computer"[3:6]
    'put'
    ```
- Strings, unlike lists, are ***immutable***--they cannot be modified

  -
    ```
    >>> comp = "Computer"
    >>> comp[3:6] = "get"
    Traceback (most recent call last):
      File "<pyshell#44>", line 1, in <module>
        comp[3:6] = "get"
    TypeError: 'str' object does not support item assignment
    ```
- However, a variable containing a string can be assigned a different value

  -
    ```
    >>> comp = comp[:3] + "get" + comp[6:]
    >>> comp
    'Comgeter'
    ```

Sets

- A *set* is an immutable, unordered collection of values, containing no duplicates
  - Because it is unordered, you cannot index into a set
  - Because it contains no duplicates, any value is either in the set once, or it isn't in the set
  - Values do not need to be all of the same type
- A set is written as a comma-separated list of values, enclosed in braces
  - **Example:** `{1, 3.6, 'a', True}`
- You can't write an empty set (a set with no elements) as `{}`, because that's a ***dictionary***--you have to write it as `set()`
- Since you can't index into a set, you have to use the simpler kind of `for` loop to process each and every element

  -
    ```
    for elem in list:
        do something with elem
    ```

Set methods

- Several set methods are given in your textbook
- You can use IDLE to explore what set methods are available, just as you can for lists
- Even better, `Help -> Python docs`, or `F1`, in IDLE will bring you to the Python documentation
  - `Language Reference` describes Python syntax and semantics
  - `Library Reference` tells you all about the available methods

Dictionaries

- A ***dictionary*** associates immutable ***keys*** with values
- **Syntax:** `{` *key1* `:` *value1*`, …,` *key_n* `:` *value_n* `}`
- You can index into a dictionary like you would a list, but the "indices" are keys, not necessarily integers
  - **Example:** `phonebook["Jane Doe"]`

Dictionary methods

- Dictionaries have methods that can be explored in IDLE, but also some with unusual syntax
  - `d[key]`
    Return the item of *d* with key *key*. Raises a `KeyError` if *key* is not in the map.
  - `d[key] = value`
    Set `d[key]` to *value*.
  - `del d[key]`
    Remove `d[key]` from *d*. Raises a `KeyError` if *key* is not in the map.
  - `key in d`
    Return `True` if *d* has a key *key*, else `False`.
  - `key not in d`
    Equivalent to `not key in d`.

Tuples

- A ***tuple*** is a way to collect a ***small*** number of related values into a single value, for convenience in handling
  - **Syntax:** `(`*value_1*`, …, `*value_n*`)`
  - **Example:**
    ```
    fun_book =("The Princess Bride",
    "William Goldman", 1973)
    ```
  - **Example:** The function `divmod(`*int_1*`, `*int_2*`)` returns a tuple containing the quotient and the remainder when *int_1* is divided by *int_2*
- Tuples are immutable
  - You can index into a tuple to retrieve values, but you can’t change any of its values

Working with tuples

- In Python, you can say: `a, b, c = 3, 5, 7`
  - This sets `a` to `3`, `b` to `5`, and `c` to `7`
  - It's the same as `(a, b, c) = (3, 5, 7)`
  - This is usually the easiest way to get information out of a tuple
- **Example:**
  ```
  (title, author, date) = fun_book
  ```

What are these all for?

- **Lists**, like `["John", "Mary", "Bill", "Mary"]`, are the most heavily used kind of collections in Python, and should be the first thing to consider
- **Sets**, like `{“John", "Mary", "Bill"}`, are just the ticket when you need to keep track of what’s “in” and what’s “out”
- **Dictionaries**, like `{"John": 5551212, "Mary": 5551234}`, are obviously for looking things up
- **Tuples**, like `("CIT 590", "Matuszek")`, are a convenient way of bundling related pieces of information together
- All this becomes a lot more obvious with experience!

The End

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

— Brian W. Kernighan and P. J. Plauger