06 Data structures

September 13, 2020

1 Introduction

A data structure is an 'object' in a program that holds a collection of data. A simple data structure might be an 'array' that holds the components of a vector, or a list of names. A more complicated data structure might represent a telephone directory, holding [name, telephone number] pairs.

Modern languages, like Python, provide a range of library (built-in) data structures. These are well-tested and optimised; it is good practice to use library data structures to make programs simpler, easier to read and more efficient.

It is possible to develop your own data structures for special purposes. In this activity we focus on selecting and using built-in data structures. The creation of your own data structures is introduced in Activity 12.

1.0.1 Objectives

- Use `list`, `tuple` and dictionary (`dict`) data structures
- Use iteratation to visit entries in a data structure
- Learn to select the right data structure for an application

2 Data structures

So far we have restricted our examples to simple built-in data types, e.g. `string`, `int`, and `float`. In practice, programs will use *data structures* to collect data together into useful packages. For example, rather than representing a vector `r` of length 3 using three floats `u`, `v` and `w`, we could represent it as a list of floats, `r = [u, v, w]`. Similarly, if we want to store the names of students in a laboratory group, rather than a string variable for each student, we could work with a list of names, e.g.:

[1]:
```
lab_group0 = ["Sarah", "John", "Joe", "Emily"]
lab_group1 = ["Roger", "Rachel", "Amer", "Caroline", "Colin"]
```

This is a much more powerful construction because we can perform operations on a list, such as checking its length (number of students in a lab group), sort the names in the list into alphabetical order, and add or remove names. We could even make a list-of-lists, e.g.

[2]:
```
lab_groups = [lab_group0, lab_group1]
```

to collect all the lab groups together into a *nested* list.

Data structures are particularly useful when passing data to functions. Say we want a function that prints the names of students in a given lab group. Rather than passing the name of each student (with different groups having differing numbers of members), we can pass just the list of lab group members to the function. Similarly, we could develop a function that computes the dot product between two vectors of arbitrary length, passing to the function just the two vectors rather than each component.

We will look at three built-in Python data structures that are commonly used. They are:

- `list`
- `tuple`
- `dict` (dictionary)

3 Lists

A `list` is a sequence of data. An 'array' in most other languages is a similar concept, but Python lists are more general than most arrays as they can hold a mixture of types. A list is constructed using square brackets:

[3]:
```python
lab_group0 = ["Sarah", "John", "Joe", "Emily", "Quang"]
print("Lab group members: {}".format(lab_group0))
print("Size of lab group: {}".format(len(lab_group0)))

print("Check the Python object type: {}".format(type(lab_group0)))
```

The function `len` returns the length (number of items) of the list.

An empty list is created by

[4]:
```python
my_list = []
```

A list of length 5 with repeated values can be created by

[5]:
```python
my_list = ["Oliver"]*5
print(my_list)
```

3.1 Iterating over lists

Looping over each item in a list (or more generally a sequence) is called *iterating*. We iterate over the members of the lab group using the syntax:

[6]:
```python
for member in lab_group0:
    print(member)
```

Say we want to iterate over the names of the lab group members, and get the position of each member in the list. We use `enumerate` for this:

[7]:
```python
for n, member in enumerate(lab_group0):
    print(n, member)
```

In the above, `n` is the position in the list and `member` is the nth entry in the list. Sometimes we need to know the position, in which case `enumerate` is helpful. However, when possible it is preferable to use 'plain' iteration. Note again that Python counts from zero - it uses zero-based indexing.

3.2 Manipulating lists

There are many functions for manipulating lists. It might be useful to sort the list:

[8]:
```python
lab_group0.sort()
for member in lab_group0:
    print(member)
```

In the above, `sort` is known as a 'method' of a `list`. It performs an *in-place* sort, i.e. `lab_group0` is sorted, rather than creating a new list with sorted entries (for the latter we would use `sorted(lab_group0)`, which returns a new list). More on methods later when we get to object-oriented design.

With lists we can add and remove students:

[9]:
```python
# Remove the second student (indexing starts from 0, so 1 is the second element)
lab_group0.pop(1)
print(lab_group0)

# Add new student "Josephine" at the end of the list
lab_group0.append("Josephine")
print(lab_group0)
```

or maybe join two groups to create one larger group:

[10]:
```python
lab_group1 = ["Roger", "Rachel", "Amer", "Caroline", "Colin"]

lab_group = lab_group0 + lab_group1
print(lab_group)
```

or create a list of group lists:

[11]:
```python
lab_groups = [lab_group0, lab_group1]
print(lab_groups)

print("---")

print("Print each lab group (name and members):")
for i, lab_group in enumerate(lab_groups):
    print(i, lab_group)
```

3.3 Indexing

Lists store data in order, so it is possible to 'index into' a list using an integer (this will be familiar to anyone who has used C), e.g.:

[12]:
```python
first_member = lab_group0[0]
third_member = lab_group0[2]
print(first_member, third_member)
```

or

[13]:
```python
for i in range(len(lab_group0)):
    print(lab_group0[i])
```

Indices start from zero, and run through to (length - 1).

Indexing can be useful for numerical computations. We use it here to compute the dot product of two vectors:

[14]:
```python
# Two vectors of length 4
x = [1.0, 3.5, 7.2, 8.9]
y = [-1.0, 27.1, 1.0, 6]

# Compute dot-product
dot_product = 0.0
for i in range(len(x)):
    dot_product += x[i]*y[i]

print(dot_product)
```

When possible, it is better to iterate over a list rather than use indexing, since there are data structures that support iterating but do not support indexing.

If we have a list-of-lists,

[15]:
```python
lab_group0 = ["Sarah", "John", "Joe", "Emily"]
lab_group1 = ["Roger", "Rachel", "Amer", "Caroline", "Colin"]
lab_groups = [lab_group0, lab_group1]
```

we can use the first index to access a list, and a second index to access the entry in that list:

[16]:
```python
group = lab_groups[0]
print(group)

name = lab_groups[1][2]
print(name)
```

```
['Sarah', 'John', 'Joe', 'Emily']
Amer
```

3.4 Heterogeneity (lists with mixed types)

Python lists are heterogeneous data structures - this means they can store mixed types, e.g.

[17]:
```python
mixed_list = ["Adam", 2 + 4j, 1.0, 4]
for entry in mixed_list:
    print(entry, type(entry))
```

```
Adam <class 'str'>
(2+4j) <class 'complex'>
1.0 <class 'float'>
4 <class 'int'>
```

Arrays in most languages are homogeneous - all types in the array must be the same.

There are *many* ways in which lists can be manipulated. The best way to learn how to perform a specific operation is to use a search engine.

3.5 List comprehension

A powerful construct in modern languages, including Python, is *list comprehension*. It is a way to succinctly build lists from other lists. It can be very useful, but should be applied sensibly as it can sometimes be difficult to read. There is an optional extension exercise at the end of this notebook that uses list comprehension.

Say we have a list of numbers and we wish to create a new list that squares each number in the original list and adds 5. Using list comprehension:

[18]:
```python
x = [4, 6, 10, 11]
y = [a*a + 5 for a in x]

print(x)
print(y)
```

```
[4, 6, 10, 11]
[21, 41, 105, 126]
```

To understand the meaning, read the statement left-to-right.

As another example, say we have a list of names and we want to

- build a new list of names that contains only the names with more than 5 characters; and
- for these names we want to add a full stop at the end.

Using list comprehension:

[19]:
```python
lab_group1 = ["Roger", "Rachel", "Amer", "Caroline", "Colin"]
group = [name + "." for name in lab_group1 if len(name) > 5]

print(lab_group1)
print(group)
```

```
['Roger', 'Rachel', 'Amer', 'Caroline', 'Colin']
['Rachel.', 'Caroline.']
```

Mastering list comprehension is not simple, but it is very powerful.

3.6 Tuples

Tuples are closely related to lists. The main difference is that a tuple cannot be changed after it has been created. In computing jargon, it is *immutable*.

For something that should not change after it has been created, such as a vector of length three with fixed entries, a tuple is more appropriate than a list. It is 'safer' in this case since it cannot be modified accidentally in a program. Being immutable ('read-only') also permits implementations to possibly exploit this to optimise for speed.

To create a tuple, use round brackets. Say at a college each student is assigned a room, and the rooms are numbered. A student 'Laura' is given room 32:

[20]:
```
room = ("Laura", 32)
print("Room allocation: {}".format(room))
print("Length of entry: {}".format(len(room)))
print(type(room))
```

```
Room allocation: ('Laura', 32)
Length of entry: 2
<class 'tuple'>
```

We can iterate over tuples in the same way as with lists,

[21]:
```
# Iterate over tuple values
for d in room:
    print(d)
```

```
Laura
32
```

and we can index into a tuple:

[22]:
```
print(room[1])
print(room[0])
```

```
32
Laura
```

We might have a student who is given permission to live outside of college. To keep track of them we still want an entry for the student, but there is no associated room number. We can create a tuple of length one using '`("a",)`', e.g.:

[23]:
```
room = ("Adrian",)
print("Room allocation: {}".format(room))
print("Length of entry: {}".format(len(room)))
print(type(room))
```

```
Room allocation: ('Adrian',)
Length of entry: 1
<class 'tuple'>
```

As part of a rooms database, we can create a list of tuples:

```
[24]: room_allocation = [("Adrian",), ("Laura", 32), ("John", 31), ("Penelope", 28),␣
       ↪("Fraser", 28), ("Gaurav", 19)]
      print(room_allocation)
```

```
[('Adrian',), ('Laura', 32), ('John', 31), ('Penelope', 28), ('Fraser', 28),
('Gaurav', 19)]
```

To make it easier for students to find their rooms on a printed room list, we can sort the list:

```
[25]: room_allocation.sort()
      print(room_allocation)
```

```
[('Adrian',), ('Fraser', 28), ('Gaurav', 19), ('John', 31), ('Laura', 32),
('Penelope', 28)]
```

We could improve the printed list by not printing those living outside of the college:

```
[26]: for entry in room_allocation:
          if len(entry) > 1:
              print("Name: {} \n  Room: {}".format(entry[0], entry[1]))
```

```
Name: Fraser
  Room: 28
Name: Gaurav
  Room: 19
Name: John
  Room: 31
Name: Laura
  Room: 32
Name: Penelope
  Room: 28
```

In summary, prefer tuples over lists when the length will not change.

4 Dictionaries (maps)

We used a list of tuples in the previous section to store room allocations. If we wanted to find which room a particular student has been allocated we would need to iterate through the list and check each name. For a very large list, this might not be very efficient.

There is a better way to do this, using a 'dictionary' (or sometimes called a 'map'). We have used indexing (with integers) into lists and tuples for direct access to a specific entry. This works if we know the index to the entry of interest. But, for a room list we identify individuals by name rather than a contiguous set of integers. Using a dictionary, we can build a 'map' from names (the *keys*) to room numbers (the *values*).

A Python dictionary (`dict`) is declared using curly braces:

[27]:
```
room_allocation = {"Adrian": None, "Laura": 32, "John": 31, "Penelope": 28,␣
 ↪"Fraser": 28, "Gaurav": 19}
print(room_allocation)
print(type(room_allocation))
```

```
{'Adrian': None, 'Laura': 32, 'John': 31, 'Penelope': 28, 'Fraser': 28,
'Gaurav': 19}
<class 'dict'>
```

Each entry is separated by a comma. For each entry we have a 'key', which is followed by a colon, and then the 'value'. Note that for Adrian we have used '`None`' for the value, which is a Python keyword for 'nothing' or 'empty'.

Now if we want to know which room Fraser has been allocated, we can query the dictionary by key:

[28]:
```
frasers_room = room_allocation["Fraser"]
print(frasers_room)
```

```
28
```

If we try to use a key that does not exist in the dictionary, e.g.

```
frasers_room = room_allocation["Frasers"]
```

Python will give an error (raise an exception). If we're not sure that a key is present, we can check:

[29]:
```
print("Fraser" in room_allocation)
print("Frasers" in room_allocation)
```

```
True
False
```

(We can also use '`in`' to check if an entry exists in a list or tuple). We can iterate over the keys in a dictionary:

[30]:
```
for d in room_allocation:
    print(d)
```

```
Adrian
Laura
John
Penelope
Fraser
Gaurav
```

or iterate over both the keys and the values:

[31]:
```
for name, room_number in room_allocation.items():
    print(name, room_number)
```

```
Adrian None
Laura 32
```

```
John 31
Penelope 28
Fraser 28
Gaurav 19
```

Note that the order of the printed entries in the dictionary is different from the input order. This is because a dictionary stores data differently from a list or tuple. Lists and tuples store entries 'linearly' in memory (contiguous pieces of memory), which is why we can access entries by index. Dictionaries use a different type of storage which allows us to perform look-ups using a 'key'.

We have used a string as the key so far, which is common. However, we can use almost any type as a key, and we can mix types. For example, we might want to 'invert' the room allocation dictionary to create a room-to-name map:

[32]:
```
# Create empty dictionary
room_allocation_inverse = {}

# Build inverse dictionary to map 'room number' -> name
for name, room_number in room_allocation.items():
    # Insert entry into dictionary
    room_allocation_inverse[room_number] = name

print(room_allocation_inverse)
```

```
{None: 'Adrian', 32: 'Laura', 31: 'John', 28: 'Fraser', 19: 'Gaurav'}
```

We can now ask who is in room 28 and who is in room 29. Not all rooms are occupied, so we should include a check that the room number is a key in our dictionary:

[33]:
```
rooms_to_check = [28, 29]

for room in rooms_to_check:
    if room in room_allocation_inverse:
        print("Room {} is occupied by {}.".format(room,␣
 ↪room_allocation_inverse[room]))
    else:
        print("Room {} is unoccupied.".format(room))
```

```
Room 28 is occupied by Fraser.
Room 29 is unoccupied.
```

5 Choosing a data structure

An important task when developing a computer program is selecting the *appropriate* data structure for a task. The more flexible the data structure, generally the less efficient it will be and it may use more memory compared to simpler data structures.

- If efficiency is not critical, pick a data structure that provides the functionality you need, and offers flexibility and ease of use. Safety should also be considered (this can be a good reason for choosing a tuple over a list).

- Use iterators rather than indexing when possible. This will allow switching from data structures that support indexing to data structures that do not support indexing (such as a '`set`' data structure).

For numerical computations, efficiency is often essential and picking the right data structure is critical for this. We will look at data structures that are specialied for numerical computations in the next notebook, and we'll see the huge difference in speed there can be between different data structures.

6 Exercises

Complete now the 06 Exercises notebook.