Object-Oriented Programming (Python)
Data Structures

Dr. Bianca Schoen-Phelan

bianca.phelan@tudublin.ie

www.biancaphelan.ie

@BSPhelan

KE-3-005a

Objectives

- Deepen understanding of types and data structures that you have used already and how they connect to general data structures
- Use behaviours

What are Data Structures

- Exactly what the name says: structures that can hold data
- Used to store collections of related data
- Python offers 4 built in data structures:
  - List
  - Tuple
  - Dictionary
  - Set
- There are other data structures that you can import, such as array

Closer Look at OOP Functionality

- You’ve encountered a lot of the built in data structures in Python already
- Now we have a closer look at their object oriented capabilities
- We will discuss when to use them instead of a normal class
- We will look at how and why extend built ins
- Three types of queues

Preview of Next Week: Objects, classes etc

- Almost everything in Python is a class, with properties and methods
- A class is a 'blueprint' in order to create an object
- Example class:

```python
class Students:
    student = 'Bianca'
```

- We can now create an object of this class:

```python
my_students = Students()
print(my_students.student)
```

[1]

The `__init__` function

- All classes have a built-in init function which is executed when a class is being initiated
- We use `init` to assign values and properties that are necessary at start up

```
class Students:
    def __init__(self, name):
        self.name = name


my_students = Students('Bianca')
print(my_students.name)
```

[1]

Object methods

- Functions that belong to an object

```
class Students:
    def __init__(self, name):
        self.name = name

    def say_my_name(self):
        print("My name is: " + self.name)


my_students = Students('Bianca')
my_students.say_my_name()
```

- `self` is a reference to the current instance of a class.
- It is used to access variables that belong to the class.

[1]

Data Structures

What are Tuples?

Tuples are immutable, but potentially changing!

- Store a number of items in order
- Immutable
- Often used either as keys to a dictionary or anywhere else where you need a hash value
- Store data not behaviour
- Typically store values that are somehow different from each other
- Primary purpose is to aggregate different pieces of data into one container
- Information can be of a different type

Tuples

- Advantage: very flexible
- Disadvantage: hard to read
- In many programs the values are not obvious, requires much scrolling to find the original tuple
- If you don't need all the values at once maybe not the right data structure

```python
stock = ("fb", 75.00, 75.03, 74.90)
print(stock[3]) #individual value
print(stock[1:3]) #slice
```

```
74.9
(75.0, 75.03)
```

[2]

Named Tuples

- What if we want to access individual elements of a tuple frequently
  - Could use an empty object, but this is only useful if you anticipate that you are going to add **behaviour** later
  - Use a Dictionary, which is most useful if you don't quite know yet what to store
  - If neither, use a named tuple

Example Named Tuple

Of course, this is also immutable.

- Constructor takes 2 arguments:
  - 1. Identifier for the named tuple
  - 2. Space separated string of attributes that the tuple can have
- It can be called like a normal class to be instantiated
- We can now access individual attributes in it as if they were an object

```
from collections import namedtuple

Stock = namedtuple("Stock", "symbol current high low")
stock = Stock("fb", 75.00, high=75.03, low=74.90)
print(stock.high)
```

```
/Users/b
75.03
```

What is a Dictionary

- If you’re tempted to use a tuple but you need to be able to change the data
- Very useful containers that allow us to map objects to each other
- An empty object with attributes are like Dictionaries (and often represented as such internally…have a look at the __dict__ attribute)
- They should be used if you want to find one object based on another object
    - Index is called the key
    - Object being stored is the value

Creating and Accessing

```
stocks = {"GOOG":(613.04, 625.10, 610.08),
          "MSFT":(30.25, 30.70, 30.19)}

print(stocks["GOOG"])
```

```
DataStructuresLecture
/Users/bianca.schoenphelan/
(613.04, 625.1, 610.08)
```

```
print(stocks["RIM"])
```

```
DataStructuresLecture
/Users/bianca.schoenphelan/
Traceback (most recent call
  File "/Users/bianca.schoe
    print(stocks["RIM"])
KeyError: 'RIM'
```

We get an error if we try to access something that does not exist.

[2]

No key error

- KeyError
  - Option to catch the key error, OR
  - Use a behaviour!!! Here: the get method.
  - Get method is very useful, optional 2nd argument writes a default if the element is not there

```
stocks = {"GOOG":(613.04, 625.10, 610.08),
          "MSFT":(30.25, 30.70, 30.19)}

print(stocks["GOOG"])
```

```
print(stocks.get("RIM"))
```

```
DataStructure
/Users/l
None
```

```
print(stocks.get("RIM", "Not found"))
```

```
DataStructuresLectur
/Users/bian
Not found
```

[2]

No key error - set a default!

- Behaves just like `get()` if the key exists
- If the key does not exist the default is entered as a value and the key-value pair is inserted into the dictionary!

```
print(stocks.setdefault("RIM", (20.00, 24.22, 30.01)))

print (stocks)
```

```
(20.0, 24.22, 30.01)
{'GOOG': (613.04, 625.1, 610.08), 'MSFT': (30.25, 30.7, 30.19), 'RIM': (20.0, 24.22, 30.01)}
```

[2]

Useful Dictionary Behaviour

Iterators can be used in loops. You've seen this example before in lists.

- `keys()`
  - Returns an iterator over all the keys in the dictionary
- `values()`
  - Returns an iterator over all the values in the dictionary
- `items()`
  - Returns an iterator over tuples (key, values) pairs for every item in the dictionary

The item() iterator

- Each key-value pair is unpacked into a variable called stock and a variable called values
- You can use any variable name here that is appropriate
- Note: Stocks don't (always) up in the same order as they've been inserted, uses a hash access for efficiency and items are therefore unsorted

```
for stock, values in stocks.items():
    print("{} last value is {}".format(stock,values[0]))
```

```
DataStructuresLecture
/Users/bianca.schoenphelan,
GOOG last value is 613.04
MSFT last value is 30.25
```

[2]

Accessing Dictionaries

- Many ways! We’ve now:
  - Square brackets as index syntax
  - get method
  - setdefaults method
  - Iterate over items
- Setting dictionary values, square brackets is easiest:

```
stocks['GOOG']=(500.63, 610.11, 596.00)

print(stocks["GOOG"])
```

```
DataStructuresLecture
/Users/bianca.schoenphelan
(500.63, 610.11, 596.0)
```

We can use the index no matter if the item exists or not.

[2]

Keys

- Previous examples all used strings as keys, but you can use any object as a key, for example
  - Numbers, or
  - Tuples, or
  - Objects
- We can also use different types of keys in the same dictionary
- But we cannot use lists
  - Lists can change at any time, so they do not hash to a value

Keys can be everything but lists

```python
randomKey_Dict = {}
randomKey_Dict["hello"]= "world"
randomKey_Dict[22] = 'twenty two'
randomKey_Dict[2.2] = 'two point two'
randomKey_Dict[('abc',123)] = 'tuples work'
```

```python
try:
    randomKey_Dict[[1,2,3]] = 'no lists'
except:
    print("no lists")
```

```python
class myDictObject:
    def __init__(self, myValue):
        self.myValue = myValue

myObject = myDictObject(30)

randomKey_Dict[myObject] = 'I like objects'
```

```python
for key,value in randomKey_Dict.items():
    print("{} has value {}".format(key, value))
```

```
no lists
hello has value world
22 has value twenty two
2.2 has value two point two
('abc', 123) has value tuples work
<__main__.myDictObject object at 0x10697fe10> has value I like objects
```

How a Lookup works

- Hash calculations!
- There is an algorithm that converts objects that are hashable into an integer value
  - Unique
  - Allows for rapid lookup
- Strings map to an integer depending on the characters in the string
- Tuples combine hashes of the items in the tuple
- Lists change their contents
  - Dictionaries cannot be used as keys to a dictionary either, same reason
- On the other side no limit to what we can use as values

Extremely Common Case: Counter

```
from collections import Counter
def letter_frequency(sentence):
    return Counter(sentence)

print(letter_frequency("hello world"))
```

```
Counter({'l': 3, 'o': 2, 'h': 1, 'e': 1, ' ': 1, 'w': 1, 'r': 1, 'd': 1})
```

The Counter behaves like a dictionary, where the keys are the items to be counted and the values are the numbers.

What are Lists

- Least object oriented Python structures
  - Unlike other programming languages lists are just available,
  - you rarely need to call methods on them, and
  - you do not need to import anything in order to use them,
  - we can also loop over a list without requesting an explicit iterator

Lists

- We covered lists extensively at the beginning of this semester
- Now we discuss when to use them as objects
- Typically used to store several instances of the same type of object
- Useful if we want to store something in some kind of order by which they were inserted, but we can sort them ourselves too
- Many other programming languages provide separate structures for queues, stacks, linked lists, array-based lists, but in Python we can use special instances of these or just use a list

Common List Behaviours

- `append(element)`
  - Adds an element to the end of a list
- `insert(index, element)`
  - Adds an element at a specific index position
- `count(element)`
  - How many times a specific element appears in the list
- `index()`
  - Tells us the specific index location of an element, raises and exception if it’s not found
- `find()`
  - Like index() but will return -1 instead of raising an exception
- `reverse()`
  - Turns the list around
- `sort()`
  - VERY INTERESTING, we discuss this in detail

sort() Behaviour

- Without an argument will place everything in a list in order
- Case sensitive!
- List of tuples will be sorted by first element in tuple
- TypeError will be raised if a list contains different types
- If we have a list of objects that we created ourselves, we should define __lt__, which means “lesser than”, to indicate a sorting order

Example of sorting a list

```
myList = ['Hello', 'Hallo', 'HELP', 'Helo']
myList.sort()
print(myList)
```

```
DataStructuresLecture
/Users/bianca.schoenphelan/PycharmPr
['HELP', 'Hallo', 'Hello', 'Helo']
```

```
myList.sort(key=str.lower)
```

```
['Hallo', 'Hello', 'Helo', 'HELP']
```

[2]

Lists

PROs

- Very versatile
- Efficient random access to all data elements in the list
- Strict ordering of elements
- Efficient support of append method

CONs

- Slow if you are inserting elements anywhere but the end, especially at the beginning of a list
- Slow for checking if an element exists in the list (doesn't scale well)
- Sorting in a particular order or reorder can be inefficient

What is a set?

- Helps to ensure that objects in a list are unique
- Sets come from mathematics and are an unordered group of (typically) unique numbers
- In Python a Set can hold any hashable object, not just numbers
    - Same objects that can be used as keys in a dictionary!
    - So no lists nor dictionaries
- NO empty sets
- Syntax very similar to dictionaries. Dictionaries separate with : if we just separate them with , it’s a set

Example of a set

- Creating a list of (song, artist) tuples
- Set of artists
- Like dictionaries, sets are also unordered

[2]

```python
songs = [("Phantom of the opera","Sarah Brightman"),
         ("Knocking on Heaven's Door", "Guns 'n Roses"),
         ("Captain Nemo", "Sarah Brightman"),
         ("November Rain", "Guns 'n Roses"),
         ("The Coffee Song", "Osibisa")]

artists = set()

for song,artist in songs:
    artists.add(artist)

print(artists)
```

```
{'Sarah Brightman', 'Osibisa', "Guns 'n Roses"}
```

Looping through the items in a set

```
for artist in artists:
    print(artist)
```

```
inOrder = list(artists)
inOrder.sort()
print(inOrder)
```

```
{"Guns 'n Roses", 'Sarah Brightman', 'Osibisa'}
true
Guns 'n Roses
Sarah Brightman
Osibisa
["Guns 'n Roses", 'Osibisa', 'Sarah Brightman']
```

[2]

Purpose of a set

- Most set behaviour relies on the existence of more than one set, examples:
    - Combine sets
    - Compare items in different sets
- Names are from maths

Set behaviour union

- Union:
  - Creates a new set out of two sets that has items from either sets
  - Duplicates show up only once
  - Like or operator
  - You can use the | operator instead of the `union()` method to do the same

Set behaviour intersection

- intersection():
  - Returns a set of items that are in both sets
  - Similar to logical & operator
  - Can use & instead of intersection() method

Set behaviour symmetric difference

- symmetric_difference():
  - What’s left
  - Set of items that are either in one set or the other but not in both

Comparing Set Behaviours

```
myArtists= {"Sarah Brightman", "Guns 'n Roses", "Osibisa", "Frank Stallone"}
friendsArtists={"Elvis Presley", "Dolly Parton", "Frank Stallone", "Sarah Brightman"}

print("All artists: {}".format(myArtists.union(friendsArtists)))
print("Both artists: {}".format(myArtists.intersection(friendsArtists)))
print("Either, but not both: {}".format(myArtists.symmetric_difference(friendsArtists)))
```

```
All artists: {'Elvis Presley', 'Frank Stallone', "Guns 'n Roses", 'Osibisa', 'Dolly Parton', 'Sarah Brightman'}
Both artists: {'Sarah Brightman', 'Frank Stallone'}
Either, but not both: {'Elvis Presley', "Guns 'n Roses", 'Osibisa', 'Dolly Parton'}
```

These methods will return the same results, no matter which set calls the other.

[2]

It matters who calls who

- issubset
- issuperset
- difference
  - Can also be represented by – operator
  - Returns all the elements in the calling set but not in the set passed as an argument

Examples

[2]

```python
myArtists= {"Sarah Brightman", "Guns 'n Roses", "Osibisa", "Frank Stallone"}
friendsArtists = {"Osibisa", "Frank Stallone"}

print("Superset: {}".format(myArtists.issuperset(friendsArtists)))
print("Superset: {}".format(friendsArtists.issuperset(myArtists)))

print("Subset: {}".format(myArtists.issubset(friendsArtists)))
print("Subset: {}".format(friendsArtists.issubset(myArtists)))

print("Difference: {}".format(myArtists.difference(friendsArtists)))
print("Difference: {}".format(friendsArtists.difference(myArtists)))
```

```
Superset: True
Superset: False
Subset: False
Subset: True
Difference: {"Guns 'n Roses", 'Sarah Brightman'}
Difference: set()
```

Set Summary

- Sets are not just containers
- They are meant to be used to operate on other sets
- Sets are also more efficient than lists when it comes to check membership with in
    - Set has the values hashed, so no matter how long the set the check with in will always take the same amount of time
    - Check with in takes longer the longer the list

Traditional Data Structures

Queues

- Very interesting data structures
- Like sets, everything a queue can do could be done by a list, but lists are not as efficient for some operations
- Example: lists don’t scale well, if you have millions of data items, list isn’t the right container for you
- 3 types of queue data structure

FIFO Queue

- First In First Out
- Most common understanding of a queue, like a line at a checkout
- Often used in concurrent applications where we have a consumer and a producer
- Good if you need to access the next element but not any element in the data structure
- Lists would be bad here as insertion at the beginning might result in shifting every item with every insert

Queue Class

- Officially infinite capacity, but often bound to a maximum
- `put()`
  - Adds an element to the back of the line
- `get()`
  - Retrieves the first element in the line
- Both have optional arguments what to do in case of a failure
  - What to do if you try to get from an empty queue or put to a full queue
  - Default is to block or wait
  - Instead can raise an exception or wait for a certain amount of time before raising an exception

Queue Class

- empty()
- full()
- Plus more methods that deal with concurrent access (not the topic of this class)

Queue Examples

- Block: access to queue is either allowed or denied until an item is available in get
- Default is **True**

[2]

```
from queue import Queue
line = Queue(maxsize=3)
line.get(block=False)
```

```
Traceback (most recent call last)
  File "/Users/bianca.schoenphela
    line.get(block=False)
  File "/Library/Frameworks/Pytho
    raise Empty
_queue.Empty
```

```
line.put("one")
line.put("two")
line.put("three")
line.put("four", timeout=1)
```

```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan
    line.put("four", timeout=1)
  File "/Library/Frameworks/Python
    raise Full
queue.Full
```

```
if line.full():
    print("yes queue is full")
```

```
yes queue is full
```

```
print(line.get())
print(line.get())
print(line.get())
print(line.empty())
```

```
one
two
three
True
```

LIFO

- Last In First Out
- Class LifoQueue
- Often called stacks (think of a stack of paper)
- Traditionally the methods on a LIFO are called push and pop
    - Not in Python
    - We use the same Queue class so we use put and get

LIFO Examples

- Block: access to queue is either allowed or denied until an item is available in get
- Default is **True**

[2]

```
from queue import LifoQueue
stack = LifoQueue(maxsize=3)
stack.put("one")
stack.put("two")
stack.put("three")
stack.put("four", block=False)
```

```
print(stack.get())
print(stack.get())
print(stack.get())
print(stack.empty())
```

```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Py
    stack.put("four", block=False)
  File "/Library/Frameworks/Python.fr
    raise Full
queue.Full
```

```
print(stack.get())
print(stack.get())
print(stack.get())
print(stack.empty())

stack.get(timeout=1)
```

Default timeout is **None**

```
three
two
one
True
```

```
True
Traceback (most recent ca
  File "/Users/bianca.sch
    stack.get(timeout=1)
  File "/Library/Framewor
    raise Empty
_queue.Empty
```

Thoughts on LIFO

- Couldn't we just use a list:
  - Yes
  - Working with the end of a list is very efficient
  - It's so efficient that LIFO uses a List under the hood
- So why does it exist:
  - If you need concurrent access, use LIFO instead of List
  - Implements a stack, so you cannot by accident insert an element at any random position

Priority Queue

- Same get() and put()
- But enforces very different behaviour to the previous ones
- Not the order but the importance of an item determines that it's to be returned
- A common convention is to store tuples in the priority queue
  - First item indicates the priority
  - Second item is the data
  - __lt__ should be implemented
- You can have multiple items with the same priority in the queue, but you get no indication on which one will be returned first
- Example use case is a product recommender system

[2]

Priority Queue Examples

```
from queue import PriorityQueue

heap = PriorityQueue(maxsize=3)
heap.put((3,"three"))
heap.put((5, "five"))
heap.put((2,"two"))
heap.put((1,"one"), block=False)
```

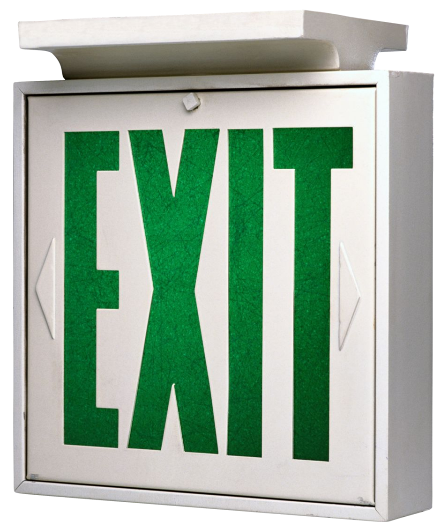
```
DataStructuresLecture
/Users/bianca.schoenphelan/PycharmProj
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Pyc
    heap.put((1,"one"), block=False)
  File "/Library/Frameworks/Python.fra
    raise Full
queue.Full
```

```
while not heap.empty():
    print(heap.get())
```

```
DataStructuresLecture
/Users/bianca.s
(2, 'two')
(3, 'three')
(5, 'five')
```

Summary

- Tuples
- Named Tuples
- Dictionaries
- Lists
- Sets
- Queues
- FIFO
- LIFO
- Priority Queue



References

1. W3 Schools Classes, https://www.w3schools.com/python/python_classes.asp, accessed Oct 2019
2. Python 3: Object Oriented Programming, Dusty Phillips, 2nd edition, 2015