COMP10001 Foundations of Computing
Objects and Types: A Closer Look
(Advanced Lecture)

Semester 2, 2016
Chris Leckie

July 31, 2016

Lecture Agenda

- This lecture:
  - Making sense of strings, lists and functions

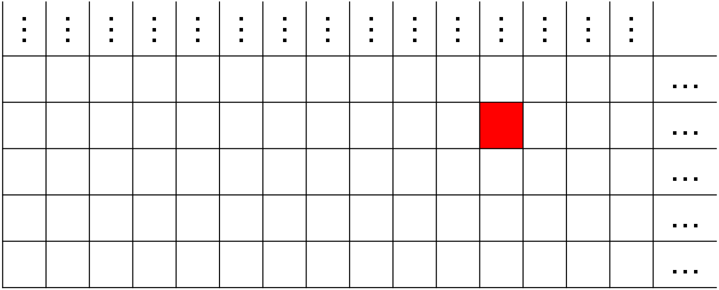
NB: All advanced lectures are unexaminable

But First ...

- Seat rearragement:
    - Move to the seat `(x,y)` defined as follows:

```
x = (age + day_of_your_birthday) % 20
y = (min(dentist_visits_in_last_year, 5) + cousins) % 10
```

- For example `(11,3)`:



Objects

- Everything in Python is an “object”, with an **unchangeable** type and a value
- There are two basic categories of type:
  - **mutable types**, where the value of the object is changeable (e.g. `list`):

    ```
    >>> lst = [0]
    >>> lst[0] = "COMP10001"
    >>> print(lst)
    ['COMP10001']
    ```

  - **immutable types**, where the value of the object is unchangeable (e.g. `int`, `str`)

Immutable Types and Nesting

- Hang on, hang on, explain the following then:

```
>>> mytup = (0, [])
>>> mytup[0] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ...
>>> mytup[1].append("hey there!")
>>> print(mytup)
(0, ['hey there!'])
```

Immutable Types and Nesting

- Hang on, hang on, explain the following then:

```
>>> mytup = (0, [])
>>> mytup[0] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ...
>>> mytup[1].append("hey there!")
>>> print(mytup)
(0, ['hey there!'])
```

- The answer is that the values of immutable types can't be changed directly, but that doesn't stop us changing the values of mutable types nested within them

Object Identity

- Internally on the computer, every object exists in "memory" and is accessible via its **identity**, defining its location in memory; this identity is also unchangeable, even for mutable types:

```
>>> lst = []
>>> id(lst)
140016396283848
>>> lst.append('newval')
>>> id(lst)
140016396283848
```

- `is` is a relational operator that checks whether two objects have the same identity

Objects and Variables I

- When we construct an object and assign it to a variable, the variable is simply assigned the **identity** of the new object; when we assign a variable to a new variable, therefore, the new variable is simply given the identity of the existing object

```
>>> int1 = 10001
>>> int2 = int1
>>> int2 is int1  # same object?
True
>>> int2 = 10001
>>> int2 is int1  # same object?
False
```

Objects and Variables II

- Although there are some optimisations in Python that confuse things slightly for immutable types:

```
>>> str1 = "woodchuck"
>>> str2 = "woodchuck"
>>> str1 == str2  # same value?
True
>>> str1 is str2  # same object?
True
>>> lst1 = []
>>> lst2 = []
>>> lst1 == lst2
True
>>> lst1 is lst2
False
```

Aside: Functions are also Objects

- All well and good, but what about functions?
- Functions are also just objects:

```
>>> type(print)
<class 'builtin_function_or_method'>
>>> id(print)
140443419192584
>>> myprint = print
>>> myprint("Hello world")
Hello world
>>> del(print)  # can only delete user-defined objects
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'print' is not defined
>>> del(myprint)
>>> print = len  # note: user-defined object
>>> print("Hello world")
11
>>> del(print)  # can delete user-defined objects
>>> print("Hello world")
Hello world
```

A Basic Pre-introduction to Dictionaries

- As powerful as lists are, if items are associated with a (unique) key, we want to be able to look them up directly, rather than search through the values in a list; dictionaries provide this, e.g.:

```
>>> lst1 = [('tim', 'duck'), ('andrew', 'dog')]
>>> for i in lst1:
...     if i[0] == 'tim':
...         print(i[1])
...
duck
>>> dict1 = {}  # initialise a new dictionary
>>> dict1['tim'] = 'duck'  # add an item
>>> dict1['andrew'] = 'dog'  # add another item
>>> print(dict1['tim'])  # look up an item
duck
```

Basic Operations

Operation	`str`	`list`	`tuple`	`dict`
Add item				
Indexing				
Lookup				
Slice				
Order preserving?				

Basic Operations

Operation	`str`	`list`	`tuple`	`dict`
Add item	—	`l.append()`	—	`d[KEY] = VAL`
Indexing				
Lookup				
Slice				
Order preserving?				

Basic Operations

Operation	str	list	tuple	dict
Add item	—	`l.append()`	—	`d[KEY] = VAL`
Indexing	`s[ID]`	`l[ID]`	`t[ID]`	`d[KEY]`
Lookup				
Slice				
Order preserving?				

Basic Operations

Operation	str	list	tuple	dict
Add item	—	`l.append()`	—	`d[KEY] = VAL`
Indexing	`s[ID]`	`l[ID]`	`t[ID]`	`d[KEY]`
Lookup	`VAL in s`	`VAL in l`	`VAL in t`	`KEY in d`
Slice				
Order preserving?				

Basic Operations

Operation	`str`	`list`	`tuple`	`dict`
Add item	—	`l.append()`	—	`d[KEY] = VAL`
Indexing	`s[ID]`	`l[ID]`	`t[ID]`	`d[KEY]`
Lookup	`VAL in s`	`VAL in l`	`VAL in t`	`KEY in d`
Slice	`s[i:j]`	`l[i:j]`	`t[i:j]`	—
Order preserving?				

Basic Operations

Operation	`str`	`list`	`tuple`	`dict`
Add item	—	`l.append()`	—	`d[KEY] = VAL`
Indexing	`s[ID]`	`l[ID]`	`t[ID]`	`d[KEY]`
Lookup	`VAL in s`	`VAL in l`	`VAL in t`	`KEY in d`
Slice	`s[i:j]`	`l[i:j]`	`t[i:j]`	—
Order preserving?	Y	Y	Y	N

Random Observations

- We can index strings, lists and tuples in constant time, irrespective of their contents and length

Random Observations

- We can index strings, lists and tuples in constant time, irrespective of their contents and length
- Lookup in a string, list or tuple gets slower as the object gets bigger, but is constant in dictionaries

Random Observations

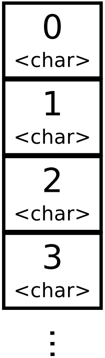
- We can index strings, lists and tuples in constant time, irrespective of their contents and length
- Lookup in a string, list or tuple gets slower as the object gets bigger, but is constant in dictionaries
- Mutation is weird! (or is it ...)

Random Observations

- We can index strings, lists and tuples in constant time, irrespective of their contents and length
- Lookup in a string, list or tuple gets slower as the object gets bigger, but is constant in dictionaries
- Mutation is weird! (or is it ...)
- Dictionary keys must be (recursively) immutable

Strings: Key Idea

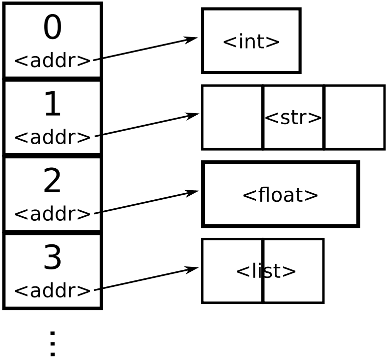
- Contiguous "array" of fixed objects (characters = `str` of length one)[a]
- Additional bookkeeping to store the length of the string (avoid overrunning the ends of the string)



[a] This is a slight over-simplification, but it gives you the basic idea

Lists: Key Idea

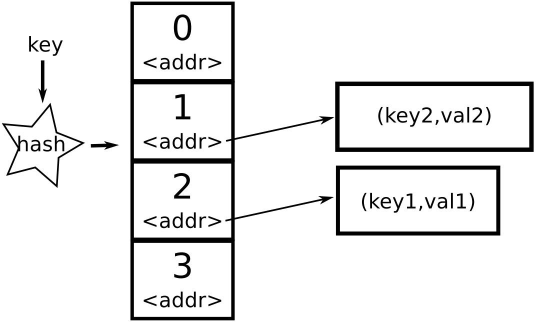
- Contiguous “array” of “pointers” to arbitrary objects[a]
- Additional bookkeeping to store the length of the list (avoid overrunning the ends of the list)



[a] Again, this is a slight over-simplification

Dictionaries: Key Idea

- Use a “hash” function to map objects onto integers with which to index a list
- Build in some mechanism to handle hash “collisions” (cf. the “birthday paradox”)



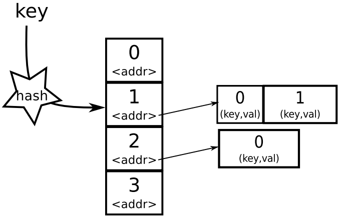
Hash Functions

- Our hash *function* must:
  - return an `int` value for all types
  - be deterministic given a key
  - be cheap
  - return a value in a given range

  and ideally should:
  - distribute keys evenly irrespective of the key source
- Our hash *table* must:
  - not use excessive storage
  - be able to handle collisions efficiently
  - support incremental additions and deletions
  - allow dynamic growth
- Hash functions in the wild?

Open Hashing

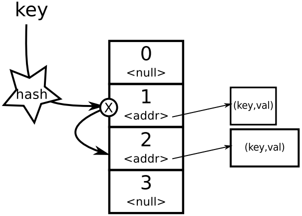
- The simplest form of a hashing is “open hashing”, where each cell in the hash table takes the form of a list, and collisions are resolved simply by appending to the end of the list



- Advantages: simple, table can’t ever “fill”
- Disadvantage: slow if lots of collisions

Closed Hashing

- A more sophisticated form of hashing with better behaviour when there is a high collision rate is "closed hashing": collisions resolved by "probing" *within hash table* for an empty cell



- Advantages: faster lookup
- Disadvantage: complexity; table can "fill up"
- Roughly what Python uses in practice

The Quirks of Assignment

- Based on what we have learned about the different types, let's see if we can make sense of the following:

```
>>> lst1 = biglist()
>>> lst2 = lst1
>>> lst1[0]
'tzRqbqAwUslkBUbkiWWAJAWBstJE_Ol'
>>> lst2[0] = -1
>>> lst1[0]
-1
>>> del(lst1)
>>> lst2[0]
-1
```

- If we think about it in terms of "pointers" (a la the internals of strings), hopefully it's less baffling

Answers to Random Observations

- We can index strings, lists and tuples in constant time, irrespective of their contents and length
  - hopefully this is less surprising now, given what we know about the implementation details

Answers to Random Observations

- We can index strings, lists and tuples in constant time, irrespective of their contents and length
  - hopefully this is less surprising now, given what we know about the implementation details
- Lookup in a string, list or tuple gets slower as the object gets bigger, but is constant in dictionaries
  - again, this is a direct function of the implementation details

Answers to Random Observations

- We can index strings, lists and tuples in constant time, irrespective of their contents and length
  - hopefully this is less surprising now, given what we know about the implementation details
- Lookup in a string, list or tuple gets slower as the object gets bigger, but is constant in dictionaries
  - again, this is a direct function of the implementation details
- Mutation is weird! (or is it ...)
  - given that variables are just pointers, hopefully considerably less weird ... verging on sensible

Answers to Random Observations

- We can index strings, lists and tuples in constant time, irrespective of their contents and length
  - hopefully this is less surprising now, given what we know about the implementation details
- Lookup in a string, list or tuple gets slower as the object gets bigger, but is constant in dictionaries
  - again, this is a direct function of the implementation details
- Mutation is weird! (or is it ...)
  - given that variables are just pointers, hopefully considerably less weird ... verging on sensible
- Dictionary keys must be (recursively) immutable
  - what would happen if we allowed the key to be mutated, in terms of its location in the dictionary?

For the Brave at Heart

- If you want to find out more about the internal implementation details of Python objects, strings and lists, the following are a good starting point:
  - `https://docs.python.org/3/reference/datamodel.html`
  - `http://www.laurentluce.com/posts/python-string-objects-implementation/`
  - `http://www.laurentluce.com/posts/python-list-implementation/`
  - `http://www.laurentluce.com/posts/python-dictionary-implementation/`

  but expect to have to go away and read up on some of the back-end algorithms

Lecture Summary

- What are objects?
- What are the basic behaviours of string, lists, tuples and dictionaries?
- What data structures underlie each?
- How does assignment work and why?
- What's the deal with mutation?
- What's the big deal about immutable keys and dictionaries?