eCornell

TOOL

Concepts in Data Structures

This tool summarizes key concepts about objects that we have covered in this course, including lists, dictionaries, and nesting. Use this document as a handy reference to refresh your understanding of objects in Python.

Programming with Objects

Class

- A custom type that is not built into Python but is instead provided by modules

Example:

```
>>> type(1)
<class ‘int’>
```

Object

- A specific value for a class type
- Sometimes referred to as an instance

Instantiation

- The act of creating an object

Constructor

- A function used to make new objects

Example:

```
>>> introcs.Point3(0,0,0)
<class 'introcs.geom.point.Point3'>(0.0,0.0,0.0)
```

Objects as Folders

Folders are used as a visual metaphor for objects. Because objects are too large to fit inside of a single variable, we think of them as folders that store variables. The variables are then referred to as attributes. The folder has a unique ID set by Python.

Example:

```
p = introcs.Point3(0,0,0)
```

id2:Point3 instance	
x	0.0
y	0.0
z	0.0

Mutable function

- Alters one of its parameters

Method

- A function inside of an object
- Not shown by visualization tools

Example:

```
p.isZero()
```

Visualizing Python Objects

Objects in Python

Every value in Python is an object and every type in Python is a class. Setting everything as an object was one of the design goals of Python 3.

Use folders as a visual representation?

Primitives: No, because they are immutable and very confusing when thought of as objects.

Tuples: Yes, because it is helpful to have a folder to be able to look at the separate slots.

0	1	2	3	4
"H"	"e"	"l"	"l"	"o"

Strings: No, because they would look exactly like tuples so we never want to represent them as folders.

More advanced types should be represented using folders because many of them are mutable and it helps to distinguish them from the primitive and basic types.

Areas of Memory

Global space:

- Location of global variables, which are any variables created by an assignment not inside of a function definition
- Lasts until you quit Python

Call stack:

- Location of call frames
- Temporary memory
- Can be thought of as scratch space

Heap space (the heap):

- Location of object "folders"
- Accessed through other variables, not directly
- Forgotten folders are periodically cleaned out

Modules and Objects

Because modules are objects, they can be used in the same way as objects. The primary difference between them is that classes can have many instances, requiring the use of the constructor function. Every module is unique. It only needs to be imported; it doesn't need a constructor function. These are sometimes referred to as singleton objects.

Functions as Variables

Employing functions as variables is very useful in processing large amounts of data. A function can be applied to every piece of data in a set. Two well-known, powerful functions that form the basis for a lot of data processing infrastructure are map() and filter().

Programming with Lists

List

- Similar to a tuple but uses square brackets instead of parentheses
- Unlike a tuple, does not need a comma for single elements
- Is mutable, allowing you to go into it and change its contents. This makes it far more advanced than a tuple. An understanding of Python memory is essential in order to use it.

List Methods

Just like tuples, lists have methods. However, they have many more because several list methods can actually alter the list.

Example:

```
>>> x = [5,6,5,9,15,23]
>>> x.append(999)
>>> x
[5, 6, 5, 9, 15, 23, 999]
```

Mutable List Functions

Because lists are mutable, you can create functions to alter them, including through small, surgical changes. This is one of the main reasons people prefer to use lists over tuples.

Lists and For-Loops

You can use an accumulator to build a list with a for-loop, just like you can a tuple. However, lists have an append method that is much more efficient than adding tuples together with +.

Tuple Example:

```
def tup_add_one(tup):
    """Returns: a copy of tup with 1
    added to every element"""
    copy = ()   # accumulator
    for x in tup:
    x = x+1
    copy = copy+(x,)    # add to end
    of copy
    return copy
```

List Example:

```
def list_add_one(lst):
    """Returns: a copy of lst with 1
    added to every element"""
    copy = []   # accumulator
    for x in tup:
    x = x+1
    copy.append(x)      # add to end
    of copy
    return copy
```

Programming with Nested Lists

Nesting

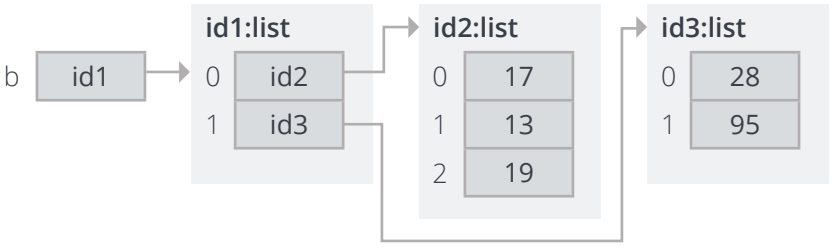
A list is able to hold other lists. This is known as a nested, or multidimensional, list.

Two-Dimensional List

- A list that contains other lists that only include primitives
- Ragged
  - The lists that are inside are not of the same length

Example:

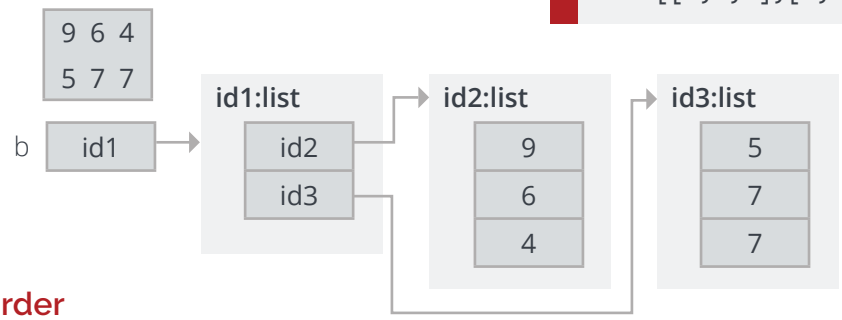
```
b = [[17,13,19],[28,95]]
```



- Rectangular (Table)
  - Contains lists all of the same length

Example:

```
b = [[9,6,4],[5,7,7]
```



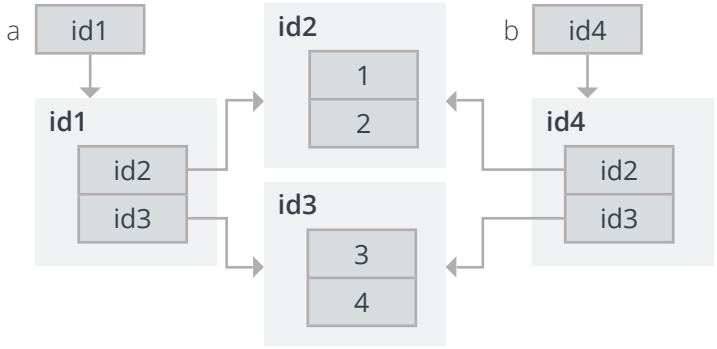
Row major order

- A table is a nested list of rows, with each row being a list of values
- This method can be used for numbers and images

(Column major order, in which a table is a nested list of columns, also exists but is limited to some scientific applications.)

Shallow copy

- Only copies the top-level list, not the rows
- Occurs when slicing a multidimensional list

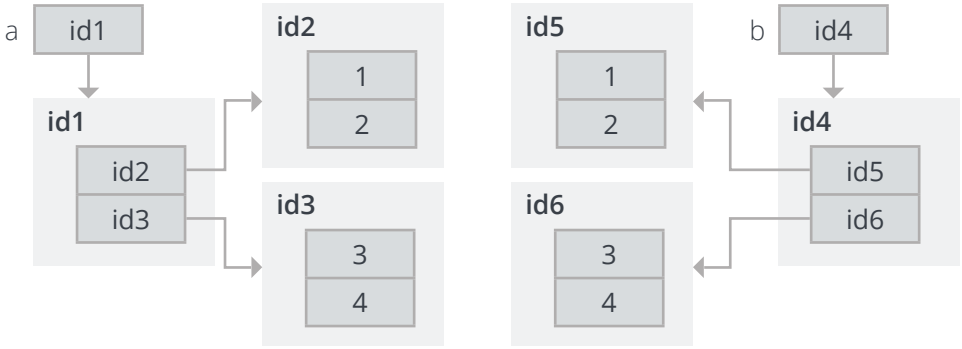


Deep copy

- Copies top-level list and all nested lists
- Usually what people want when they copy

Example:

```
>>> import copy
>>> a = [[1,2],[2,3]]
>>> b = a[:]               # b is a shallow copy
>>> c = copy.deepcopy(a)   # c is a deep copy
```



Function Design on Two-Dimensional Lists

For immutable functions, follow the same basic steps as you would with a standard list: go over the nested lists with a for-loop and use an accumulator to gather the results. However, there are two important differences:

- Multiple for-loops are often needed for nested lists. For a two-dimensional list, two loops are needed, one for each dimension.
- Multiple accumulators, such as one accumulator for each for-loop, may be needed in some cases.

Designing a mutable function on a nested list also closely follows the design steps on a standard list. The looping is not over the list but over the range of positions. No accumulator or return statement is needed. The one big difference, however, is that multiple for-loops may be needed.

Programming with Dictionaries

Dictionary

- One of the most important types
- Shares similarities with a list, including mutability
- Built of key-value pairs:
  - A key is a unique identifier
  - A value is any non-unique Python value

Example:

```
d = { 'jrs1': 'John', 'jrs2': John',
'wmw2': Walker' }
```

Dictionaries can be changed in three ways:

- Reassigning values
- Adding new keys
- Deleting keys

Dictionaries and Objects

Aside from using bracket notation instead of dot notation, dictionaries are similar to objects in several ways. However, one key difference is that objects are less flexible but safer. Objects often display error messages telling you when you're doing something wrong. Dictionaries are more flexible but also more dangerous to use, as you can sometimes get your data wrong.

Nesting dictionaries

Because values can be anything, dictionaries can be nested like lists.

Example:

```
d = { 'a': [1,2], 'b': [3,4] }
d = { 'a': { 'c':1, 'd':2}, 'b': {
'e':3, 'f':4}}
```

Immutable Functions

Design of immutable functions on dictionaries is very similar to doing so on lists. The process is to go over the dictionary with a for-loop, then use an accumulator to gather the results. One difference involves keys and values. When using a for-loop with a dictionary, you only get the keys. To access the values, you need to combine the bracket notation with the for-loop.

Mutable Functions

Designing mutable functions on dictionaries is done differently than on lists. It is okay to loop over a dictionary that you want to change because you're looping over the keys, not the values. However, you may never add or remove keys inside of the body of the for-loop for a dictionary. This causes an infinite loop with a list, but with a dictionary this will cause Python to crash and fail.