Python Review

Jay Summet

IPRE

2005-12-31

Outline

- Compound Data Types:
  Strings, Tuples, Lists & Dictionaries
- Immutable types:
  - Strings
  - Tuples
- Accessing Elements
- Cloning Slices
- Mutable Types:
  - Lists
  - Dictionaries
- Aliasing vs Cloning

Compound Data Types - A data type in which the values are made up of elements that are themselves values.

- Strings – Enclosed in Quotes, holds characters, (immutable):
  “This is a String”
- Tuples – Values separated by commas, (usually enclosed by parenthesis) and can hold any data type (immutable):
  (4 , True, “Test”, 34.8)
- Lists – Enclosed in square brackets, separated by commas, holds any data type (mutable):
  [4, True, “Test”, 34.8]
- Dictionaries – Enclosed in curly brackets, elements separated by commas, key : value pairs separated by colons, keys can be any immutable data type, values can be any data type:
  { 1 : “I”, 2 : ”II”, 3 : “III”, 4 : “IV”, 5 : “V” }

Immutable Types

- Strings and Tuples are immutable, which means that once you create them, you can not change them.
- The assignment operator (=) can make a variable point to strings or tuples just like simple data types:
  myString = “This is a test”
  myTuple = (1,45.8, True, “String Cheese”)
- Strings & Tuples are both *sequences*, which mean that they consist of an ordered set of elements, with each element identified by an index.

Accessing Elements

```
myString = “This is a test.”
```

- You can access a specific element using an integer *index* which counts from the front of the sequence (starting at ZERO!)

```
myString[0] produces 'T'
myString[1] produces 'h'
myString[2] produces 'i'
myString[3] produces 's'
```

- The `len()` function can be used to find the length of a sequence. Remember, the last element is the length minus 1, because counting starts at zero!

```
myString[ len(myString) – 1] produces '.'
```

Counting Backwards

`myString = “This is a test.”`

- As a short cut (to avoid writing `len(myString)`) you can simply count from the end of the string using negative numbers:

`myString[ len(myString) – 1 ]`	produces '.'
`myString[-1]`	also produces '.'
`myString[-2]`	produces 't'
`myString[-3]`	produces 's'
`myString[-4]`	produces 'e'

Index out of Range!

```
myString = “This is a test.”
```

- Warning! If you try to access an element that does not exist, Python will throw an error!

  `myString[5000]` produces An ERROR!

  `myString[-100]` produces An ERROR!

Traversal

MyString = “Daddy, I want a Peanut butter and Jelly bread.”

How many “J”s are in the string?

Traversals

- Many times you need to do something to every element in a sequence. (e.g. Check each letter in a string to see if it contains a specific character.)
- Doing something to every element in a sequence is so common that it has a name: a Traversal.
- You can do a traversal using a while loop as follows:

```
index = 0
while ( index < len (myString) ):
   if myString[index] == 'J':
      print “Found a J!”
   index = index + 1
```

Easy Traversals – The FOR Loop

- Python makes traversals easy with a FOR loop:

```
for ELEMENT_VAR in SEQUENCE:
    STATEMENT
    STATEMENT
```

- A for loop automatically assigns each element in the sequence to the (programmer named) ELEMENT_VAR, and then executes the statements in the block of code following the for loop once for each element.
- Here is an example equivalent to the previous WHILE loop:

```
for myVar in myString:
      if myVar == 'J':
         print “I found a J!”
```

- Note that it takes care of the indexing variable for us.

Grabbing Slices from a Sequence

- The slice operator will clip out part of a sequence. It looks a lot like the bracket operator, but with a colon that separates the "start" and "end" points.

```
SEQUENCE_VAR [ START : END ]
myString = "This is a test."
```

`myString[0:2]`	produces	'Th'	(0, 1, but NOT 2)
`myString[3:6]`	produces	's I'	(3-5, but NOT 6)

POP QUIZ:

`myString[ 1: 4 ]`	produces	?
`myString[ 5: 7 ]`	produces	?

Slices – Default values for blanks

```
SEQUENCE_VAR [ START : END ]
myString = “This is a test.”
```

- If you leave the “start” part blank, it assumes you want zero.

```
myString[ : 2]        produces 'Th'    (0,1, but NOT 2)
myString[ : 6]        produces 'This i' (0-5, but NOT 6)
```

- If you leave the “end” blank, it assumes you want until the end of the string

```
myString[ 5 : ]  produces 'is a test.'    (5 – end)

myString [ : ]   produces 'This is a test.' (0-end)
```

Using Slices to make clones (copies)

- You can assign a slice from a sequence to a variable.
- The variable points at a copy of the data.

```
myString = “This is a test.”

hisString = myString [ 1 : 4 ]

isString = myString [ 5 : 7 ]

testString = myString [10 : ]
```

Tuples

- Tuples can hold any type of data!
- You can make one by separating some values with comas. Convention says that we enclose them with parenthesis to make the beginning and end of the tuple explicit, as follows:

  ```
  (4, True, “Test”, 14.8)
  ```
- NOTE: Parenthesis are being overloaded here, which make the commas very important!
  - (4) is NOT a tuple (it's a 4 in parenthesis)
  - (4,) IS a tuple of length 1 (note the trailing comma)
- Tuples are good when you want to group a few variables together (firstName, lastName)   (x,y)

Using Tuples to return multiple pieces of data!

- Tuples can also allow you to return multiple pieces of data from a function:

```
def area_volume_of_cube( sideLength):
  area = 6 * sideLength * sideLength
  volume = sideLength * sideLength * sideLength
  return  area, volume



myArea, myVolume = area_volume_of_cube( 6 )
```

- Note that in this example we left out the (optional) parenthesis from the tuple (area,volume)!
- You can also use tuples to swap the contents of two variables: `a,b = b,a`

Tuples are sequences!

- Because tuples are sequences, we can access them using the bracket operator just like strings.

```
myTuple = ( 4,48.8,True, “Test”)
```

`myTuple[0 ]`	produces4
`myTuple[ 1 ]`	produces48.8
`myTuple[ 2 ]`	producesTrue
`myTuple[ -1 ]`	produces'Test'

“Changing” Immutable data types

- Immutable data types can not be changed after they are created. Examples include Strings and Tuples.
- Although you can not change an immutable data type, you can create a new variable that has the changes you want to make.
- For example, to capitalize the first letter in this string:

```
myString = “all lowercase.”
myNewString = “A” + myString[1:]
```

- We have concatenated two strings (a string with the capital letter A of length 1, and a clone of myString missing the first, lowercase, 'a').

Changing mutable data types

- Mutable data types can be changed after they are created. Examples include Lists and Dictionaries.
- These changes happen “in place”.

```
myList = ['l', 'o', 'w', 'e', 'r', 'c', 'a', 's',
  'e']
myList[0] = 'L'
```

`print myList` produces:

```
[ 'L', 'o', 'w', 'e', 'r', 'c', 'a', 's',
  'e']
```

Lists – like strings & tuples, but mutable!

- Lists are a mutable data type that you can create by enclosing a series of elements in square brackets separated by commas. The elements do not have to be of the same data type:

  ```
  myList = [ 23, True, 'Cheese”, 3.1459 ]
  ```

- Unlike Strings and tuples, individual elements in a list can be modified using the assignment operator. After the following commands:

```
myList[0] = True
myList[1] = 24
myList[3] = “Boo”
```

myList contains: `[ True, 24, 'Cheese', 'Boo' ]`

Different Elements, Different Types

- Unlike strings, which contain nothing but letters, list elements can be of any data type.

```
myList = [ True, 24, 'Cheese', 'Boo' ]

for eachElement in myList:
   print type(eachElement)
```

produces:

```
<type 'bool'>
<type 'int'>
<type 'str'>
<type 'str'>
```

List Restrictions and the append method

- Just like a string, you can't access an element that doesn't exist. You also can not assign a value to an element that doesn't exist.
- Lists do have some helper methods. Perhaps the most useful is the append method, which adds a new value to the end of the list:

```
myList = [ True, 'Boo', 3]
myList.append(5)
print myList
[ True, 'Boo', 3, 5]
```

String to list

```
>>># convert into a list of char's
>>> w = list(s)
>>> w
['b', 'i', 'o', 'v', 'i', 't', 'r', 'u',
'm']

>>> s = 'a few words'
>>> w = s.split()
>>> w
['a', 'few', 'words']
```

Converting lists between strings

```
>>> s = 'biovitrum'   # create a string

>>> w = list(s)       # convert into a list of char's

>>> w
['b', 'i', 'o', 'v', 'i', 't', 'r', 'u', 'm']

>>> w.reverse()

>>> w

['m', 'u', 'r', 't', 'i', 'v', 'o', 'i', 'b']

>>> r = ''.join(w)    # join using empty string

>>> r
'murtivoib'

>>> d = '-'.join(w)   # join using dash char

>>> d
'm-u-r-t-i-v-o-i-b'
```

Pop Quiz:

convert roman letters to numbers

I

II

III

IV

V

VI

VII

VIII

IX

X

And now, for something completely different!

- Dictionaries are an *associative* data type.
- Unlike sequences, they can use ANY immutable data type as an index.
- Dictionaries will associate a key (any immutable data), with a value (any data).

Dictionary example: Accessing

```
arabic2roman = { 1 : “I”, 2 : ”II”, 3 :
  “III”, 4 : “IV”, 5 : “V” }
```

- You can retrieve a value from a dictionary by indexing with the associated key:
  `arabic2roman[1]` produces 'I'
  `arabic2roman[5]` produces 'V'
  `arabic2roman[4]` produces 'IV'

Reassigning & Creating new Key/Value associations

```
arabic2roman = { 1 : “I”, 2 : ”II”, 3 :
  “III”, 4 : “IV”, 5 : “V” }
```

- You can assign new values to existing keys:

  `arabic2roman [1] = 'one'`

  now:

  `arabic2roman[1]` produces 'one'

- You can create new key/value pairs:

  `arabic2roman[10] = 'X'`

  now

  `arabic2roman[10]` produces 'X'

Dictionaries and default values with the get method

- If you use an index that does not exist in a dictionary, it will raise an exception (ERROR!)
- But, you can use the get(INDEX, DEFAULT) method to return a default value if the index does not exist. For example:

```
arabic2roman.get(1,”None”)
               produces    'I'
arabic2roman.get(5, “None”)
               produces    'V'
arabic2roman.get(500, “None”)
               produces    'None'
arabic2roman.get(“test”, “None”)
               produces    'None'
```

Difference Between Aliases & Clones

- More than one variable can point to the same data. This is called an Alias (or a reference).
- For example:
  ```
  a = [ 5, 10, 50, 100]
  b = a
  ```

-
  ```
  # b = a[:]
  ```
  Now, a and b both point to the same list.

- If you make a change to the data that a points to, you are also changing the data that b points to:
  ```
  a[0] = 'Changed'
  ```

- a points to the list: [“Changed”, 10, 50, 100]
- But because b points to the same data, b also points to the list ['Changed', 10, 50, 100]

Cloning Data with the Slice operator

- If you want to make a clone (copy) of a sequence, you can use the slice operator ([:])
- For example:

  ```
  a = [ 5, 10, 50, 100]
  b = a[0:2]
  ```

  Now, b points to a cloned slice of a that is [5, 10]

- If you make a change to the data that a points to, you do NOT change the slice that b points to.

  ```
  a[0] = 'Changed'
  ```

- a points to the list: [“Changed”, 10, 50, 100]
- b still points to the (different) list [5, 10]

Cloning an entire list

- You can use the slice operator with a default START and END to clone an entire list (make a copy of it)
- For example:

  ```
  a = [ 5, 10, 50, 100]
  b = a[: ]
  ```

  Now, a and b point to different copies of the list with the same data values.

- If you make a change to the data that a points to, nothing happens the the list that b points to.

  ```
  a[0] = 'Changed'
  ```

- a points to the list: [“Changed”, 10, 50, 100]
- b points to the the copy of the old a: [5, 10, 50, 100]

Be very careful!

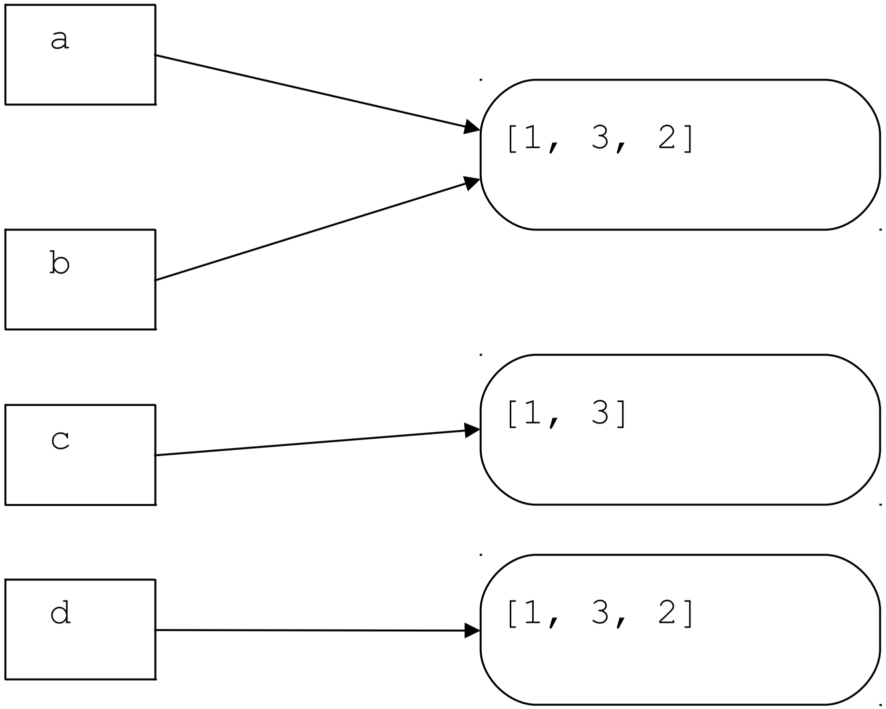
- The only difference in the above examples was the use of the slice operator in addition to the assignment operator:
- `b = a`
  vs
  `b = a[:]`
- This is a small difference in syntax, but it has a very big semantic meaning!
- Without the slice operator, a and b point to the same list. Changes to one affect the other.
- With it, they point to different copies of the list that can be changed independently.

Objects, names and references

A variable is a name referencing an object
An object may have several names referencing it
Important when modifying objects in-place!
You may have to make proper copies to get the effect you want
For immutable objects (numbers, strings), this is never a problem

```
>>>  a  =  [1,  3,  2]
>>>  b  =  a
>>>  c  =  b[0:2]
>>>  d  =  b[:]
```

```
>>>  b.sort()
>>>#  'a'  is  affected!
>>>  a
[1,  2,  3]
```



Jay Summet