Python 101 (part 4): Feeding The Snake

By Vikram Vaswani

Table of Contents

Building Blocks

In the previous article, I examined Python's control flow routines, which consist primarily of the "while" and "for" loops, and which, correctly used, allow you to do some pretty fancy things with your Python program. Combine that with Python's lists, and you have a pretty potent combination...

However, that's just the tip of the iceberg. In addition to numbers, strings and lists, Python also offers two other very interesting variants, referred to in Python−lingo as dictionaries and tuples. Similar in concept to lists − both allow you to group related items together − these two data structures possess their own distinctive properties and methods, and provide clever Python programmers with a great deal of power and flexibility.

Over the course of this article, I'm going to explore these two structures in greater detail. Keep reading.

Running The Lights

I'll begin with tuples, which share most of their properties and methods with lists. In fact, they even look alike − take a look:

```
Python 1.5.2 (#1, Aug 25 2000, 09:33:37) [GCC 2.96 20000731
(experimental)] on
linux-i386
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> CreepyCrawlies = ("spiders", "ants", "lizards")
>>> CreepyCrawlies
('spiders', 'ants', 'lizards')
>>>
```

As you can see, a tuple is another "container" variable, which can contain one or more values. In the example above, "CreepyCrawlies" is a tuple containing the values "spiders", "ants" and "lizards".

Unlike lists, which are enclosed within square braces, tuples are enclosed within parentheses, with values separated by commas. The various elements of the tuple are accessed via an index number, with the first element starting at zero. So, to access the element "spiders", you would use the notation

```
>>> CreepyCrawlies[0]
'spiders'
>>>
```

while

```
>>> CreepyCrawlies[2]
'lizards'
>>>
```

− essentially, the tuple name followed by the index number enclosed within square braces. You'll remember this "zero-based indexing" from lists.

Defining a tuple is simple − simply assign values (enclosed in square braces) to a variable, as illustrated below:

```
>>> pasta = ("macaroni", "spaghetti", "lasagne", "fettucine")
>>>
```

The rules for choosing a tuple name are the same as those for any other Python variable − it must begin with a letter, and can optionally be followed by more letters and numbers. Like a list, a tuple can mix different types of elements − the following example creates a tuple containing strings, numbers, lists and even another tuple.

```
>>> allMixedUp = ("ola", 67, "pink fox, yellow moon",
43534.57, [4,
"four"], ("red", "blue", "green"))
>>> allMixedUp
('ola', 67, 'pink fox, yellow moon', 43534.57, [4, 'four'],
('red', 'blue',
'green'))
>>> allMixedUp[0]
'ola'
>>> allMixedUp[2]
'pink fox, yellow moon'
>>> allMixedUp[4]
[4, 'four']
>>> allMixedUp[4][1]
'four'
>>> allMixedUp[5][2]
'green'
>>>
```

It should be noted that when Python sees two or more comma−separated elements, it automatically creates a tuple for them...which means that the following code snippets are equivalent.

```
>>> lights = "red", "green", "blue"
>>> type(lights)

>>> lights = ("red", "green", "blue")
>>> type(lights)

>>>
```

That said, it's a good idea to always explicitly enclose your tuples within parentheses; this avoids confusion when you review your code seven years later, and also makes it easier to read.

The type() function above is used to find out the type of a specific variable − whether string, list, tuple or little green hybrid.

Strange Food

Tuples can be concatenated with the + operator,

```
>>> CreepyCrawlies
('spiders', 'ants', 'lizards')
>>> pasta
('macaroni', 'spaghetti', 'lasagne', 'fettucine')
>>> strangeFood = CreepyCrawlies + pasta
>>> strangeFood
('spiders', 'ants', 'lizards', 'macaroni', 'spaghetti',
'lasagne',
'fettucine')
>>>
```

and repeated with the * operator.

```
>>> pasta * 5
('macaroni', 'spaghetti', 'lasagne', 'fettucine', 'macaroni',
'spaghetti',
'lasagne', 'fettucine', 'macaroni', 'spaghetti', 'lasagne',
'fettucine',
'macaroni',
'spaghetti', 'lasagne', 'fettucine', 'macaroni', 'spaghetti',
'lasagne',
'fettucine')
>>>
```

"Slices" of a tuple can be extracted using notation similar to that used with lists − take a look:

```
>>> pasta = ("macaroni", "spaghetti", "lasagne", "fettucine",
"tagliatelle", "rigatoni")
>>> pasta[0]
'macaroni'
>>> pasta[2:]
('lasagne', 'fettucine', 'tagliatelle', 'rigatoni')
>>> pasta[1:3]
```

```
('spaghetti', 'lasagne')
>>> pasta[:4]
('macaroni', 'spaghetti', 'lasagne', 'fettucine')
>>> pasta[-4]
'lasagne'
>>> pasta[-1]
'rigatoni'
>>>
```

The built-in len() function can be used to calculate the number of elements in a tuple,

```
>>> pasta
('macaroni', 'spaghetti', 'lasagne', 'fettucine',
'tagliatelle', 'rigatoni')
>>> len(pasta)
6
>>>
```

while the "in" and "not in" operators can be used to test for the presence of a particular element in a tuple. A match returns 1 (true), while a failure returns 0 (false).

```
>>> pasta
('macaroni', 'spaghetti', 'lasagne', 'fettucine',
'tagliatelle', 'rigatoni')
>>> "macaroni" in pasta
1
>>> "ravioli" in pasta
0
>>> "ravioli" not in pasta
1
>>> "ravio" not in pasta
1
>>> "maca" in pasta
0
>>>
```

Unbreakable

As you can see, tuples share many of their properties and methods with lists. However, they differ in one very important area: lists are mutable, while tuples are immutable (like strings). It is not possible to alter tuple elements once a tuple has been created; the only way to accomplish this is to create a new tuple containing the new elements.

```
>>> listSample = ["red", "green", "blue"]
>>> listSample[2] = "pink"
>>> listSample
['red', 'green', 'pink']
>>> tupleSample = ("red", "green", "blue")
>>> tupleSample[2] = "pink"
Traceback (innermost last):
File "", line 1, in ?
TypeError: object doesn't support item assignment
>>>
```

A new tuple can be created either by adding two or more existing tuples,

```
>>> CreepyCrawlies
('spiders', 'ants', 'lizards')
>>> pasta
('macaroni', 'spaghetti', 'lasagne', 'fettucine')
>>> strangeFood = CreepyCrawlies + pasta
>>> strangeFood
('spiders', 'ants', 'lizards', 'macaroni', 'spaghetti',
'lasagne',
'fettucine')
>>>
```

or by extracting slices from existing tuples and joining them into a new structure.

```
>>> veryStrangeFood = CreepyCrawlies[1], pasta[3], pasta[5]
>>> print veryStrangeFood
('ants', 'fettucine', 'rigatoni')
>>>
```

Or, if you want to be really fancy, you could use the list() and tuple() functions to convert a tuple into a list,

```
>>> pasta
('macaroni', 'spaghetti', 'lasagne', 'fettucine',
'tagliatelle', 'rigatoni')
>>> type(pasta)

>>> # convert to list
>>> pastaList = list(pasta)
>>> type(pastaList)

>>> print (pastaList)
['macaroni', 'spaghetti', 'lasagne', 'fettucine',
'tagliatelle', 'rigatoni']
>>> pastaList[5]=""
>>> print (pastaList)
['macaroni', 'spaghetti', 'lasagne', 'fettucine',
'tagliatelle', '']
>>>
```

and vice−versa.

```
>>> lights = ["red", "green", "blue"]
>>> type(lights)

>>> # convert to tuple
>>> lightsTuple = tuple(lights)
>>> type(lightsTuple)

>>> print lightsTuple
('red', 'green', 'blue')
>>>
```

Finally, you can iterate through a tuple in much the same way as a list − with the "for" loop.

```
>>> pasta
```

```
('macaroni', 'spaghetti', 'lasagne', 'fettucine',
'tagliatelle', 'rigatoni')
>>> for x in pasta:
... print "I like", x
...
I like macaroni
I like spaghetti
I like lasagne
I like fettucine
I like tagliatelle
I like rigatoni
>>>
```

At this point, you're probably wondering why you need tuples when you already have lists to contend with. It's pretty simple: the built−in immutability of tuples provides an automatic defense against program code that attempts to change tuple elements, thereby ensuring the integrity of the data contained within this structure. As we progress through this tutorial, you'll see examples of where this might come in useful.

Keys To The Kingdom

One of the significant features of lists and tuples is that the values stored within them can only be accessed via a numerical index. The implication of this is obvious: if you need to access an element of the list or tuple, you need to first know its exact location in the structure. Since this can get complicated with large lists, Python offers you a simpler way to access list values, using a data structure known as a dictionary.

Dictionaries are similar to lists and tuples, in that they allow you to group related elements together. However, where lists and tuples use an index to identify and locate specific elements or groups of elements, dictionaries use keywords ("keys") to accomplish the same task. In fact, Python dictionaries are the equivalent of Perl hashes or PHP associative arrays.

Let's take a look at a simple example to demonstrate this:

```
>>> characters = {"new hope":"Luke", "teacher":"Yoda", "bad
guy":"Darth"}
>>>
```

Dictionaries are typically enclosed within curly braces, and each element within a dictionary consists of a "key" and a "value" associated with that key, separated from each other by a colon (:). Since every value in the dictionary is associated with a specific key, you can reference a specific value via its key.

```
>>> characters["bad guy"]
'Darth'
>>> characters["new hope"]
'Luke'
>>>
```

Defining a dictionary is not very difficult − simply assign key−value pairs (enclosed in curly braces) to a variable, as illustrated below:

```
>>> characters = {"new hope":"Luke", "teacher":"Yoda", "bad
guy":"Darth"}
>>>
```

You can also assign elements one at a time to an empty dictionary,

```
>>> opposites = {}
>>> opposites
{}
>>> opposites["white"] = "black"
>>> opposites["yes"] = "no"
>>> opposites["day"] = "night"
>>> opposites
{'day': 'night', 'white': 'black', 'yes': 'no'}
>>>
```

or add new elements to an existing one.

```
>>> characters = {"new hope":"Luke", "teacher":"Yoda", "bad
guy":"Darth"}
>>> characters["princess"] = "Leia"
>>> characters["worse guy"] = "The Emperor"
>>> characters
{'princess': 'Leia', 'teacher': 'Yoda', 'new hope': 'Luke',
'bad guy':
'Darth',
'worse guy': 'The Emperor'}
>>>
```

As you can see, the elements in a dictionary are not placed in a specific sequence, as with lists and tuples; new elements are assigned to random places within the dictionary space.

It should be noted that keys and values in a dictionary need not be restricted to strings alone − numbers and tuples serve equally well as keys (which must be immutable),

```
>>> mishmash = {"jack":"jill", 70:"sixty-nine plus one",
(2,2):4,
(1,2,3):(3,2,1)}
>>> mishmash["jack"]
'jill'
>>> mishmash[70]
'sixty-nine plus one'
```

```
>>> mishmash[(2,2)]
4
>>> type(mishmash[(2,2)])

>>> mishmash[(1,2,3)]
(3, 2, 1)
>>> type(mishmash[(1,2,3)])

>>>
```

---

while values could include strings, numbers, and even other dictionaries, lists or tuples (isn't it cool how Python allows you to nest one data structure within another?)

```
>>> moremush = {
... "Friends":["Rachel", "Ross", "Joey", "Phoebe", "Monica",
"Chandler"],
... "icecream":{"strawberry":"pink", "chocolate":"brown",
"vanilla":"white"}
... }
>>> # first value is list, second is nested dictionary
>>> moremush
{'icecream': {'chocolate': 'brown', 'vanilla': 'white',
'strawberry':
'pink'}, 'Friends': ['Rachel', 'Ross', 'Joey', 'Phoebe',
'Monica',
'Chandler']}
>>>
>>> moremush["icecream"]
{'chocolate': 'brown', 'vanilla': 'white', 'strawberry':
'pink'}
>>> type(moremush["icecream"])

>>> moremush["icecream"]["strawberry"]
'pink'
>>> type(moremush["icecream"]["strawberry"])

>>> type(moremush["Friends"])

>>> moremush["Friends"][0]
'Rachel'
>>>
```

Looking Up The Dictionary

Unlike tuples, dictionaries are a "mutable" object type, which means that the elements contained within them can be changed at will.

As you've already seen, adding a new element to a dictionary is as simple as assigning a value to a key.

```
>>> characters = {"new hope":"Luke", "teacher":"Yoda", "bad
guy":"Darth"}
>>> characters["princess"] = "Leia"
>>> characters["worse guy"] = "The Emperor"
>>> characters
{'princess': 'Leia', 'teacher': 'Yoda', 'new hope': 'Luke',
'bad guy':
'Darth',
'worse guy': 'The Emperor'}
>>>
```

You can also use this technique to update existing dictionary elements with new values,

```
>>> characters["teacher"] = "Obi-Wan"
>>> characters
{'princess': 'Leia', 'teacher': 'Obi-Wan', 'new hope': 'Luke',
'bad guy':
'Darth', 'worse guy': 'The Emperor'}
>>>
```

or use the del() method to remove individual key-value pairs from the dictionary.

```
>>> del (characters["worse guy"])
>>> characters
{'princess': 'Leia', 'teacher': 'Obi-Wan', 'new hope': 'Luke',
'bad guy':
'Darth'}
>>>
```

A built−in update() method makes it possible to copy elements from one dictionary to another.

```
>>> charactersCopy = {}
>>> charactersCopy.update(characters)
>>> charactersCopy
{'teacher': 'Obi−Wan', 'princess': 'Leia', 'new hope': 'Luke',
'bad guy':
'Darth'}
>>>
```

while a clear() method allows you to empty the container of all elements,

```
>>> charactersCopy.clear()
>>> charactersCopy
{}
>>>
```

The built−in len() function can be used to calculate the number of key−value pairs in a dictionary.

```
>>> characters
{'princess': 'Leia', 'teacher': 'Obi−Wan', 'new hope': 'Luke',
'bad guy':
'Darth'}
>>> len (characters)
4
>>>
```

It should be noted at this point that dictionaries do not support concatenation or repetition, and, since they don't use numerical indices, it's not possible to extract slices of a dictionary either. In fact, attempting this will cause Python to barf all over your screen,

```
>>> characters
{'princess': 'Leia', 'teacher': 'Obi−Wan', 'new hope': 'Luke',
'bad guy':
```

```
'Darth'}
>>> opposites
{'day': 'night', 'white': 'black', 'yes': 'no'}
>>> newDict = characters + opposites
Traceback (innermost last):
File "", line 1, in ?
TypeError: bad operand type(s) for +
>>>
```

---

as will the deadly sin of trying to access a key which doesn't exist.

```
>>> characters
{'princess': 'Leia', 'teacher': 'Obi-Wan', 'new hope': 'Luke',
'bad guy':
'Darth'}
>>> characters["robot"]
Traceback (innermost last):
File "", line 1, in ?
KeyError: robot
>>>
```

As a matter of fact, you can use the has_key() built-in method to check whether or not a particular key exists, thereby making it easier to avoid errors like the one above.

```
>>> characters
{'princess': 'Leia', 'teacher': 'Obi-Wan', 'new hope': 'Luke',
'bad guy':
'Darth'}
>>> characters.has_key("teacher")
1
>>> characters.has_key("princess")
1
>>> characters.has_key("robot")
0
>>>
```

Of Keys And Locks

Two of the most important and useful methods associated with Python dictionaries are the keys() and values() methods, used to return a list of the keys and values stored in a dictionary respectively. The next example should make this clearer:

```
>>> characters
{'princess': 'Leia', 'teacher': 'Obi-Wan', 'new hope': 'Luke',
'bad guy':
'Darth'}
>>> characters.keys()
['princess', 'teacher', 'new hope', 'bad guy']
>>> characters.values()
['Leia', 'Obi-Wan', 'Luke', 'Darth']
>>>
```

Since dictionaries do not use numeric indices, it would not normally be possible to iterate through a dictionary using a "for" loop. However, since both keys() and values() return a list (which *is* compatible with "for"), they can come in extremely handy. Take a look at the following Python program, which sets up a dictionary mapping years to director names, and then iterates through the dictionary with a "for" loop.

```
#!/usr/bin/python

# define a dictionary
directors = {1995:"Mel Gibson", 1996:"Anthony Minghella",
1997:"James
Cameron", 1998:"Steven Spielberg", 1999:"Sam Mendes",
2000:"Steven
Soderbergh"}

# loop through
for x in directors.keys():
print "And the Oscar for Best Director in", x, "goes to",
directors[x]
```

Here's the output.

```
And the Oscar for Best Director in 1999 goes to Sam Mendes
And the Oscar for Best Director in 1998 goes to Steven
Spielberg
And the Oscar for Best Director in 1997 goes to James Cameron
And the Oscar for Best Director in 1996 goes to Anthony
Minghella
And the Oscar for Best Director in 1995 goes to Mel Gibson
And the Oscar for Best Director in 2000 goes to Steven
Soderbergh
```

In case you'd like it in chronological order, you can always use the sort() function on the list.

```
#!/usr/bin/python

# define a dictionary
directors = {1995:"Mel Gibson", 1996:"Anthony Minghella",
1997:"James
Cameron", 1998:"Steven Spielberg", 1999:"Sam Mendes",
2000:"Steven
Soderbergh"}

# get and sort the list
years = directors.keys()
years.sort()

# loop through
for x in years:
print "And the Oscar for Best Director in", x, "goes to",
directors[x]
```

In this case, I've first obtained a list of keys, sort()ed them, and then used the sorted list in the "for" loop.

As opposed to keys() and values(), the items() method returns something a little more complex − a list of tuples, each tuple containing a key−value pair from the dictionary. It's actually quite elegant − take a look!

```
>>> characters
{'princess': 'Leia', 'teacher': 'Obi−Wan', 'new hope': 'Luke',
'bad guy':
'Darth'}
>>> characters.items()
```

```
[('princess', 'Leia'), ('teacher', 'Obi-Wan'), ('new hope',
'Luke'), ('bad
guy', 'Darth')]
>>>
```

And that's about it for today. While this article was not as code-intensive as the previous ones (hey, even you need a break!), we've nevertheless broken a lot of new ground. You now know about two of Python's more unusual data structures, the tuple and the dictionary, have a fair idea of the capabilities and features of each, and will be able to impress your friends with your exhaustive knowledge of Best Director Oscar-winners.

In the next article, we'll be moving away from built-in Python data structures to something a little more interesting: interacting with files on the system. Python can be used to read data from, and write data to, files on your filesystem - and I'll be giving you all the gory details next time. Make sure you come back for that!