Data Structures in Python

1. Tuples

Prof. Moheb Ramzy Girgis
Department of Computer Science
Faculty of Science
Minia University

Tuples, Lists and Dictionaries

- In addition to basic data types that store numerical values and strings, Python defines three data types for storing more complex data:
  - The ***list:*** a sequence of related data
  - The ***tuple:*** a list whose elements may not be modified
  - A ***dictionary:*** a list of values that are accessed through their associated keys.
- These data types are high-level implementations of simple data structures that enable Python programmers to manipulate many types of data quickly and easily.
- Some Python modules (e.g., **Cookie** and **cgi**) use these data types to provide simple access to their underlying data structures.

Tuples

- Tuples are similar to lists, except tuples are ***immutable***.
- The following program compares the usage of lists versus tuples:

```
my_list = [1, 2, 3, 4, 5, 6, 7] # Make a list
my_tuple = (1, 2, 3, 4, 5, 6, 7) # Make a tuple
print('The list:', my_list) # Print the list
print('The tuple:', my_tuple) # Print the tuple
# Access an element
print('The first element in the list:', my_list[0])
print('The first element in the tuple:', my_tuple[0])
print('All the elements in the list:', end=' ')
for elem in my_list: # Iterate over the elements of a list
    print(elem, end=' ')
print()
print('All the elements in the tuple:', end=' ')
for elem in my_tuple: # Iterate over the elements of a tuple
    print(elem, end=' ')
print()
```


Tuples

```
print('List slice:', my_list[2:5]) # Slice a list
print('Tuple slice:', my_tuple[2:5]) # Slice a tuple
print('Try to modify the first element in the list . . .')
my_list[0] = 9 # Modify the list
print('The list:', my_list)
print('Try to modify the first element in the tuple . . .')
my_tuple[0] = 9 # Is tuple modification possible?
print('The tuple:', my_tuple)
```

- We see that this program does not run to completion.
- The next to the last statement in the program:

  my_tuple[0] = 9

  generates a run-time exception because *tuples are immutable*.
- *Once we create tuple object, we cannot change that object's contents*.

Tuples

Output

```
The list: [1, 2, 3, 4, 5, 6, 7]
The tuple: (1, 2, 3, 4, 5, 6, 7)
The first element in the list: 1
The first element in the tuple: 1
All the elements in the list: 1 2 3 4 5 6 7
All the elements in the tuple: 1 2 3 4 5 6 7
List slice: [3, 4, 5]
Tuple slice: (3, 4, 5)
Try to modify the first element in the list . . .
The list: [9, 2, 3, 4, 5, 6, 7]
Try to modify the first element in the tuple . . .
Traceback (most recent call last):
  File "C:/Python/My Python Programs/tupleTest.py", line 21, in
<module>
    my_tuple[0] = 9 # Is tuple modification possible?
builtins.TypeError: 'tuple' object does not support item assignment
```


Tuples

- The following table compares lists to tuples.

Feature	List	Tuple
Mutability	mutable	immutable
Creation	`lst = [i, j]`	`tpl = (i, j)`
Element access	`a = lst[i]`	`a = tpl[i]`
Element modification	`lst[i] = a`	*Not possible*
Element addition	`lst += [a]`	*Not possible*
Element removal	`del lst[i]`	*Not possible*
Slicing	`lst[i:j:k]`	`tpl[i:j:k]`
Slice assignment	`lst[i:j] = []`	*Not possible*
Iteration	`for elem in lst:...`	`for elem in tpl:...`

- Unlike with lists, we *cannot modify* an element within a tuple, we *cannot add* elements to a tuple, *nor can we remove* elements from a tuple.
- If we have a variable assigned to a tuple, we always can reassign that variable to a different tuple.

Tuples

- Such an assignment simply binds the variable to a different tuple object—it does not modify the tuple to which the variable originally was bound.
- The parentheses are optional in the following statement:

  my_tuple = (1, 2, 3)

  The following statement is equivalent:

  my_tuple = 1, 2, 3
- Lists can hold heterogeneous data types, and so too can tuples:

  t = (2, 'Fred', 41.2, [30, 20, 10])
- In general practice, however, many Python programmers favor storing only homogeneous types in lists and prefer tuples for holding heterogeneous types.

Tuples

➢ **Tuple unpacking:**

- The following code unpacks a tuple into separate variables:

  t = 3, 'A', 99

  val, letter, quant = t
- After this code executes *val* will refer to 3, *letter* will be assigned to the string 'A', and *quant* will be another name for the integer 99.
- Tuple unpacking is convenient if you need to extract most, if not all, of the elements from a tuple.
- If you need only one element from a potentially large tuple, the index operator is a better choice, as shown here:

  t = 3, 'A', 99

  letter = t[1]

  Here letter is assigned to the string 'A'.

Tuples

- If you wish to extract several components from a tuple and ignore others, tuple extraction with ***"throw away" _ variables*** can be a good choice.
- The following code illustrates this:

```
t = 3, 'A', 99, 16, 0, 42
_, letter, _, _, quant, rating = t
```

- The nameless _ variable will end up with the value 16, but this variable is meant to be ignored.
- The code that follows these statements would be interested only in the variables *letter*, *quant*, and *rating*.

➢ **Converting a *tuple* to a *list* using the *list function*:**

- Example:

```
>>> tpl = 1, 2, 3, 4, 5, 6, 7, 8
>>> tpl
(1, 2, 3, 4, 5, 6, 7, 8)
>>> list(tpl)
[1, 2, 3, 4, 5, 6, 7, 8]
```


Tuples

➢ **Converting a *list* to a *tuple* using the *tuple function*:**

- Example:

```
>>> lst = ['a', 'b', 'c', 'd']
>>> lst
['a', 'b', 'c', 'd']
>>> tuple(lst)
('a', 'b', 'c', 'd')
```

- Neither the ***list*** nor ***tuple function*** actually modifies its argument; that is, ***tuple(lst)*** does not modify *lst*, and ***list(tpl)*** does not modify *tpl*.
- The ***list function*** makes a new list out of the contents of a tuple, and the ***tuple function*** makes a new tuple out of the elements in a list.

Tuples

- **Generating a sequence of tuples from two lists using the *zip function*:**
- Example:

```
>>> lst1 = [1, 2, 3, 4, 5, 6, 7, 8]
>>> lst2 = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> for t in zip(lst1, lst2):
---         print(t)
---
(1, 'a')
(2, 'b')
(3, 'c')
(4, 'd')
(5, 'e')
(6, 'f')
(7, 'g')
(8, 'h')
```

- The Python *zip function* pairs up elements from two different sequences.
- If one of the sequences is shorter than the other, the zip function stops at the shorter sequence and ignores the remainder of the longer sequence.

Tuples

- The paired-up elements are tuples, and the sequences can be lists or sequences constructed from generators.
- **Example:** The following program constructs a sequence of tuples with their first elements derived from a list and their second elements obtained from a generator. Note that the generator's sequence is shorter than the list.

```
def gen(n):
    """ Generates the first n perfect squares, starting with zero:
    0, 1, 4, 9, 16,..., (n - 1)^2. """
    for i in range(n):
        yield i**2
for p in zip([10, 20, 30, 40, 50, 60], gen(4)):
    print(p, end=' ')
print()
```

Run

(10, 0) (20, 1) (30, 4) (40, 9)

- The *zip function* does not return a list; like *range*, it returns an object over which we can iterate.

Tuples

- We can make a list from a ***zip object*** using the ***list*** conversion function:

```
>>> list(zip(range(5), range(10, 0, -1)))
[(0, 10), (1, 9), (2, 8), (3, 7), (4, 6)]
```

- We can use the ***zip function*** and ***list comprehension*** to build elaborate lists.
- Suppose we wish to make a new list from two existing lists.
  - The first element in our new list will be the sum of the first elements from the two original lists.
  - Similarly, the second element in our new list will be the sum of the second elements in the two original lists, and so forth.
- We can use ***zip*** to pair up the elements, as the following interactive sequence illustrates:

Tuples

```
>>> for p in zip([1, 2, 3, 4, 5], [10, 11, 12, 13, 14]):
...     print(p)
...
(1, 10)
(2, 11)
(3, 12)
(4, 13)
(5, 14)
```

- We want to add together the components of each tuple. To print each sum we could write:

```
>>> for (x, y) in zip([1, 2, 3, 4, 5], [10, 11, 12, 13, 14]):
...     print(x+y)
...
11
13
15
17
19
```

Tuples

- We can reassemble these pieces into a *list comprehension* to build our list of sums:

```
>>> [x + y for (x, y) in zip([1, 2, 3, 4, 5], [10, 11, 12, 13, 14])]
[11, 13, 15, 17, 19]
```

- When treated as a Boolean expression, the empty tuple (()) is interpreted as ***False***, and any other tuple is considered ***True***.
- Since they are so similar, why does Python have both lists and tuples?
- Under some circumstances an executing program can perform optimizations on immutable objects that would be impossible with mutable objects.
- These optimizations can increase the program's performance.

Arbitrary Argument Lists

- If we need a function to be flexible enough to add two or three numbers, we can implement such a function with default arguments.
- The following program illustrates such a function:

```
def sum(a, b, c=0):
    return a + b + c
print(sum(3, 4))
print(sum(3, 4, 5))
```

Run

```
7
12
```

- A function that is capable of adding up to five numbers is equally easy:

```
def sum(a, b=0, c=0, d=0, e=0):
    return a + b + c + d + e
```

- When we define a function we specify the individual parameters it accepts, providing default values as needed.
- In the function definitions we have seen to this point the number of parameters is fixed.

Arbitrary Argument Lists

- We need a way to define a function in such a way so that it can accept an arbitrary number of parameters.
- Fortunately Python has a mechanism for doing this.
- The following program illustrates how to write such a function.

```
def sum(*nums):
        print(nums)             # See what nums really is
        s = 0 # Initialize sum to zero
        for num in nums:        # Consider each argument passed to the function
                s += num        # Accumulate their values
        return s                # Return the sum
print(sum(3, 4))
print(sum(3, 4, 5))
print(sum(3, 3, 3, 3, 4, 1, 9, 44, -2, 8, 8))
```

Run

```
(3, 4)
7
(3, 4, 5)
12
(3, 3, 3, 3, 4, 1, 9, 44, -2, 8, 8)
84
```

- Here the ***sum*** function handles as many actual parameters as the User can provide.

Arbitrary Argument Lists

- The single asterisk (*) before the formal parameter *nums* indicates the parameter is a collection of values.
- As the output shows, the formal parameter *nums* is a ***tuple*** wrapping all the actual parameters sent by the caller.
- Since *nums* is simply a ***tuple***, we can iterate over it with ***for statement*** to extract all the actual parameters provided by the caller.
- A function definition may contain at most one of these parameters that represents a tuple of arguments, and, if present, this parameter must appear after all the named, single formal parameters, if any.
- In the following ***sum*** function, callers must provide at least two parameters but may pass more:

```
def sum(num1, num2, *extranums):
    s = num1 + num2
    for n in extranums:
            s += n
    return s
```

Note that the formal parameters *num1* and *num2* must appear before **extranums* in ***sum's*** formal parameter list.

Arbitrary Argument Lists

- Python supports a concept known as ***generalized unpacking***.
- It extends simple unpacking by using the ***asterisk*** to represent a collection of elements not covered by individual variables during the unpacking.
- The following interactive sequence shows how *generalized unpacking* can extract a prefix of a tuple, storing the remainder of the tuple's elements in a list:

```
>>> a = 1, 2, 3, 4, 5, 6, 7, 8
>>> a
(1, 2, 3, 4, 5, 6, 7, 8)
>>> x, y, *rest = a
>>> x
1
>>> y
2
>>> rest
[3, 4, 5, 6, 7, 8]
```

Note that, the elements in the remainder of the tuple are copied into a ***list***, not a ***tuple***

19

Arbitrary Argument Lists

- The following interactive sequence shows how *generalized unpacking* can extract a suffix of a tuple:

```
>>> a = 1, 2, 3, 4, 5, 6, 7, 8
>>> a
(1, 2, 3, 4, 5, 6, 7, 8)
>>> *start, x, y = a
>>> start
[1, 2, 3, 4, 5, 6]
>>> x
7
>>> y
8
```

- At most one * expression may appear in a generalized unpacking expression.
- The next sequence unpacks a prefix and a suffix of the elements:

```
>>> a = 1, 2, 3, 4, 5, 6, 7, 8
>>> a
(1, 2, 3, 4, 5, 6, 7, 8)
>>> x1, x2, *middle, x3, x4 = a
>>> x1
1
>>> x2
2
>>> middle
[3, 4, 5, 6]
>>> x3
7
>>> x4
8
```

20

Arbitrary Argument Lists

- Generalized unpacking works with lists as well:

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8]
>>> x1, x2, *midlist, x3, x4 = a
>>> x1
1
>>> x2
2
>>> midlist
[3, 4, 5, 6]
>>> x3
7
>>> x4
8
>>> x, y, z = ['a', 'b', 'c']
>>> x
'a'
>>> y
'b'
>>> z
'c'
```

- The ***print function*** knows how to handle a list argument, and it also accepts a variable number of arguments.
- Consider the following program which demonstrates the power of the **** unpacking*** operator for flexible list printing:

Arbitrary Argument Lists

```
lst = [2*i for i in range(6)]
# Typical list printing
print(lst)
# Print just the list elements
print(*lst)
# Print the list in a special way
print(*lst, sep=" and ", end="--that's all folks!\n")
```

Run

```
[0, 2, 4, 6, 8, 10]
0 2 4 6 8 10
0 and 2 and 4 and 6 and 8 and 10--that's all folks!
```

- We can unpack a tuple directly from a range object without the need to involve a for statement:

```
>>> x, y, z = range(10, 31, 10)
>>> x
10
>>> y
20
>>> z
30
```

You must ensure the number of values match exactly the number of variables.