02-1 Tuples

CSI 500

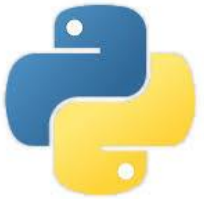
Spring 2018

Course material derived from:
Downey, Allen B. 2012. "Think Python, 2nd Edition". O'Reilly Media Inc., Sebastopol CA.

"How to Think Like a Computer Scientist" by Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers. Oct 2012
http://openbookproject.net/thinkcs/python/english3e/index.html

Tuples are immutable

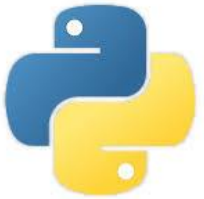
- Useful for holding small sets of values
  - Kind of like yellow-sticky pad notes
- Can form using comma separated list of values
  - e.g.: t = 'a', 'b', 'c', 'd', 'e'
- Can form using parenthesis too
  - e.g.: t = ('a', 'b', 'c', 'd', 'e')
- Single-element tuples require final comma
  - e.g.: t = 'a', # this makes a tuple
- But note difference here
  - e.g.: x = ('a') # this assigns x to 'a'
- can also use built-in tuple() function
  - e.g: t = tuple('abcde') # makes a tuple



```
# let's make a tuple

t = 'a', 'b', 'c'          # one way

t
('a', 'b', 'c')

t = tuple('abc')           # or another way
t
('a', 'b', 'c')

type(t)                    # tuple is a built-in type
<class 'tuple'>

len(t)      # total number of tuple elements
3

t[1]        # access using index operators []
'b'
```

Tuple assignment

- Tuples are immutable, and can't be individually updated
  - how is this helpful?
  - kind of like Strings: you make new ones out of old ones
- tuples provide an elegant way to assign multiple values simultaneously
  - makes logic flow cleaner



```
# here's a common idom: swap
a = 5
b = 7
tmp = 0

# do the swap like a normal computer scientist
tmp = a
a = b
b = tmp

# do it using tuples instead
a, b = b, a

# here's another idiom: split a string variable
addr = 'monty@python.org'
uname, domain = addr.split('@')
uname
'monty'
domain
'python.org'
```

Variable length argument tuples

- Functions can **gather** variable numbers of arguments
  - a parameter beginning with * will 'gather' arguments into a tuple
- Opposite of gather is **scatter**
  - takes a tuple beginning with * and explodes it into single elements
  - similar syntax using the * operator

```
# let's gather arguments into a tuple
def printall( *args ):
        print(args)

printall( 1, 2.0, 'a')
(1, 2.0, 'a')           # returns a tuple


# let's scatter a tuple into elements
t = (1, 2.0, 'a')

print( t )              # print the tuple
(1, 2.0, 'a')

print( *t )             # print the scattered tuple
1 2.0 a
```

Lists and tuples

- zip is a built in function that takes 2 or more sequences and returns a set of tuples
  - matches the elements one-by-one
  - called 'zip' as reference to interleaved metal teeth of a zipper
  - often used with a for loop to iterate
  - returns a set of tuples
- Often used with lists
  - zip return values put into a list
  - iterate over each list element (tuple)
  - multiple indexes can be used - handy for iterating (see 'letter' 'number' example)

```
# use the zip function
a = 'abc'                # a is a string
t = [0, 1, 2]            # t is a list
zip( a, t )
<zip object at 0x000001F998F0FBC8>

for pair in zip( a, t ):
    print( pair )

('a', 0)
('b', 1)
('c', 2)

p = list( zip(a, t) )
P
[('a', 0), ('b', 1), ('c', 2)]

for letter, number in p:
    print('letter = ', letter, ' number = ', number)

letter =  a  number =  0
letter =  b  number =  1
letter =  c  number =  2
```

Lists and tuples (2)

- combining zip, tuples, and for loops gives a common idiom for traversing sequences
  - allows nice way to pairwise compare lists
- you can also use 'enumerate' if you need to access numerical index values
  - automatically keeps track of which index you're using

```
# idiom for pairwise comparing any sequences
def has_match( t1, t2 ):
        for x, y in zip( t1, t2):
                if x == y:
                        return True
        return False

has_match( tuple('abc'), tuple('def'))
False       # no common letters
has_match( tuple('abc'), tuple('axy'))
True        # at least one common letter

has_match( [1,2,3], [2,3,5] )
False       # no common number
has_match( [1,2,3], [7, 2, 8])
True        # at least one common number

# use 'enumerate' to keep track of index numbers
for index, element in enumerate('abc'):
        print(index, element)

0 a
1 b
2 c
```

Sequences of sequences

- In many contexts, sequences can be used interchangeably - which is best?
  - when generating function return values, tuples work well
  - when using a sequence as a dictionary key, you must use an immutable type like a string or a tuple
  - when passing values to a function, an immutable tuple reduces chance for accidental aliasing behavior
- Tuples are immutable, so can't use sort or reverse to change in place
  - however, you can use the "sorted" and "reversed" methods to return new sorted list or reversed tuples object

```
# let's sort (of) sort a tuple
t = tuple('xyzabc')
t
('x', 'y', 'z', 'a', 'b', 'c')
t.sort()
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    t.sort()
AttributeError: 'tuple' object has no attribute 'sort'

sorted(t)
['a', 'b', 'c', 'x', 'y', 'z']

for ch in reversed( t ):
    print(ch)


c
b
a
z
y
x
```

Summary

- tuples are a built-in Python data type
  - immutable - can't be changed once created
  - tuple elements are comma separated set of values enclosed in parenthesis
  - values may be any type
- tuples have some handy functions
  - len() gives number of tuple elements
  - zip() and enumerate() for iterating over groups of tuples
  - sorted() and reversed() for processing tuples