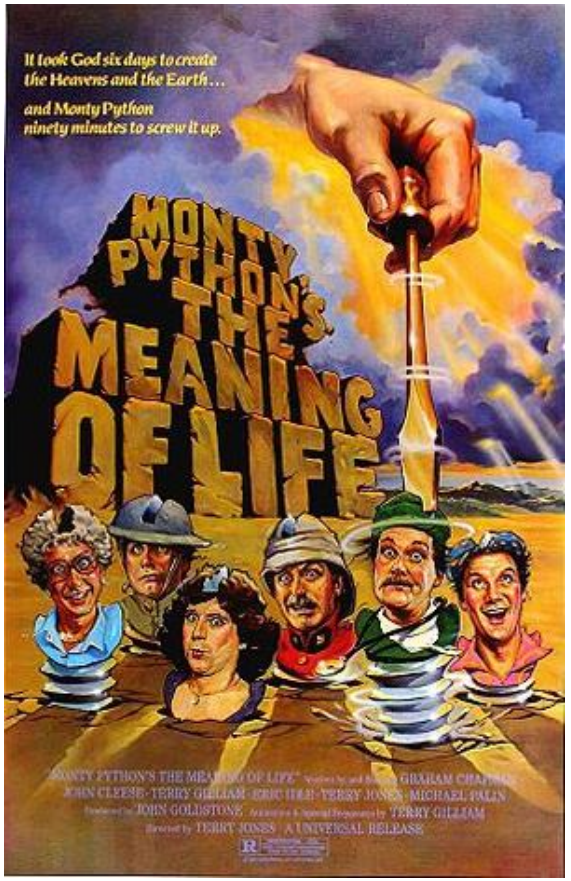
Tuples

- Assignment
- Return
- Variable length
- Comparison
- Sequences of sequences



Tuples

- A tuple is a sequence of values of any type
  - Indices have to be integer
  - Lot like lists, except they are *immutable*
- A tuple is a comma-separated list of values
  - and if just one, put a comma at the end
- Examples:

```
>>>t = 'a',  'b', 'c', 'd', 'e'
>>>t1 = ('a', 'b', 'c', 'd', 'e')
>>>t2 = 'a',
>>>type (t2)
<type 'tuple')
```

- Not a tuple

```
>>>t3 = ('a')
>>>type(t3)
<type 'str'>
```

Creating a tuple

- The function `tuple()` creates a tuple with no items
  - This is the name of a built-in function, do NOT use it as variable name

```
>>>t = tuple()
>>> print (t)
()
```

- If argument to function `tuple()` is a sequence (string, list, tuple)
  - result is tuple with the elements of the sequence

```
>>>t = tuple('parrot')
>>> print (t)
('p', 'a', 'r', 'r', 'o', 't')
```

operators

- Most list operators also work on tuples
  - bracket operators indexes an element

```
>>>t = 'a',  'b', 'c', 'd', 'e'
>>>print (t[0])
'a'
```

- slice operator selects a range

```
>>>print (t[1:3])
('b', 'c')
```

Tuples are immutable…

- Can’t change one of its elements…

```
>>>t[0] = ‘A’
  TypeError: object doesn’t support item assignment
```

- Work around to replace one tuple with another using slices

```
>>>t = (‘A’,) + t[1:]
>>>print (t)
(‘A’, ‘b’, ‘c’, ‘d’, ‘e’)
```

Tuple assignment

- Most conventional assignments might require a temporary variable
- Example: swap

```
>>>temp = a
>>>a = b
>>>b= temp
```

- Convenient way to do in python
  - left side is tuple of variable, right is tuple of expressions
  - number on both sides have to match

```
>>> a, b = b, a


>>> a, b = 1, 2, 2
ValueError: too many values to unpack
```

Assignment

- The right side can be any kind of sequence (string, list or tuple)

```
>>>addr = 'catherine.Stringfellow@mwsu.edu'
>>> uname, domain = addr.split('@')
```

- return value from split is a list with two elements
  - the first is assigned to uname, second to domain.

```
>>>print (uname)
catherine.Stringfellow
>>>print (domain)
mwsu.edu
```

Returning tuples

- Strictly speaking a function can only return one value
  - So if you need to return more, put it in a tuple
  - Example: have divmod return both the quotient and the remainder

```
def divmod(x, y):
return x//y, x%y

t = divmod(7, 3)
print (t)
    → (2, 1)
```

  - or, store elements separately

```
>>>quot, rem = divmod(7, 3)
>>>print (quot, rem)
 2  1
```

Variable-length argument tuples

- Functions can have a variable number of arguments…we know that
  - can use * to gather arguments into a tuple
  - use any id you want, but args conventional

  - Example:

```
def printall(*args):
    print (args)

>>>printall (1, 2.0, '3')
(1, 2.0, '3')
```

Variable-length argument tuples

- Can scatter tuple into arguments using * operator
  - Example:

```
>>>t = (7, 3)
>>>divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

  - Correct way:

```
>>>t = (7, 3)
>>>divmod(*t)
(2, 1)
```

Variable-length argument tuples

- Many built-in functions use variable-length argument tuples.
  - For example `max` and `min` do.

```
>>>max (1, 3, 5, 3)
3
```

  - But `sum` does not.

```
>>>sum(1, 3, 5, 3)
TypeError: sum expected at most 2 arguments, got 3
```

Lists and tuples

- zip is a built-in function – takes 2+ sequences and ‘zips’ them into a list of tuples
  - Each tuple has one element from each sequence
  - Python 3: zip returns an iterator of tuples, but an iterator behaves like a list
  - Example :

```
>>>s = ‘abc’
>>>t = [0, 1, 2]
>>>zips (s,t)
[(‘a’, 0), (‘b’, 1), (‘c’, 2)]
```

  - if one sequence shorter, then result has length of shorter, rest are ignored

Traverse list of tuples

- Traverse a list of tuples

```
t=[('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print (number, letter)
```

- will give you

```
0 a
1 b
2 c
```

Combine with zip

- Can traverse two (or more sequences) at the same time
- Example:
  - This will return True if there is an index i such that t1[i] == t2[i]

```
def has_match(t1, t2):
    for x, y in zip(t1,t2):
        if x == y:
            return True
    return False
```

enumerate()

- Need to traverse the elements of a sequence and their indices, you can use the built-in function enumerate
- Example:

```
for index, element in enumerate('abc'):
    print (index, element)
```

- Output:

```
0 a
1 b
2 c
```

- NOTE: for each element in the sequence,

  a tuple is produced (index, element)

Dictionaries and tuples

- Dictionaries have a method called `items()` that returns a list of tuples (in no particular order)
  - We've already seen this

```
>>>d = {'a':0, 'b':1, 'c':2}
>>>t = d.items()
>>>print (t)
[('a', 0), ('c', 2), ('b', 1)]
```

- In Python 3, `items()` returns an iterator (but they behave like lists)

Dictionaries and tuples

- You can also use a list of tuples to initialize a new dictionary
  - We've already seen this

```
>>>t = [('a', 0), ('c', 2), ('b', 1)]
>>>d = dict(t)
>>>print (d)
{'a':0, 'b':1, 'c':2}
```

- Combining dict() with zip() gives a concise way to create a dictionary

```
>>>d = dict(zip('abc', range(3)))
>>>print (d)
{'c': 2, 'a': 0, 'b': 1}
```

update()

- update() also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary

```
d = dict(zip('abc', range(3)))
print (d)
d.update([('d', 5), ('e', 9)])
print (d)
```

- will give us

```
{'b': 1, 'c': 2, 'a': 0}
{'b': 1, 'e': 9, 'c': 2, 'a': 0, 'd': 5}
```

Combining

- Combining `items`, tuple assignment and `for`, you can traverse the keys and values of a dictionary
  - seen before

```
for key, val in d.items():
    print (val, key)
```

- will give us

```
0 a
2 c
1 b
5 d
9 e
```

Tuples as keys in dictionaries

- Can't use list as key, but can use tuple
  - Example: map last_name, first_name pairs to telephone numbers

    ```
    directory[last,first] = number
    ```
  - Expression in brackets is a tupel.
  - Can now use tuple assignment to traverse this dictionary

    ```
    for last, first in directory:
        print (first, last, directory[last, first]
    ```
  - will give us

    ```
    Catherine Stringfellow 1-940-397-4578
    John Cleese 1-865-350-3459
    ```

    etc.

Comparing tuples

- Relational operators work with tuples and other sequences
  - Python compares corresponding elements
    - if equal goes on to next element, until it finds elements that differ

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>>(0, 1, 200000) < (0, 3, 4)
True
```

  - `sort()` function works the same way
    - see demo of sortwordsbylength.py

Sequences of Sequences

- These lecture focused on lists of tuples
- BUT you can have lists of lists, tuples of tuples, tuples of lists, lists of dictionaries, dictionaries with strings as keys and values, etc., etc.,
- So just refer to all of these as sequences of sequences
  - Most of the time, a function or operator will work on all of them
  - If immutable, just don't put as a lvalue in an assignment statement

Sequences of Sequences

- When might you prefer tuples?
  - in a return statement, syntactically easier to create a tuple than a list
  - If you want to use a sequence as a key, you must use an immutable type like a tuple or a string (can't use a list)
  - Passing as an argument, using tuples reduces potential problems due to aliasing
- Tuples are immutable, so they don't have methods like `sort` and `reverse`, which modify existing lists
  - but we have built-in functions like `sorted` and `reversed`, which takes a sequence as parameter and returns a new list with same elements in the desired ordered