>>> conjugations = {
    "see": ["saw", "sees"],
    "walk": ["walked", "walks"],
    "do": ["did", "does"],
    "be": ["was", "is"]
}

>>> conjugations["see"]

>>> conjugations["walk"]

>>> conjugations["walk"][1]

>>> conjugations["walk"][1][0]

>>> [word[0] for word in conjugations["be"]]

>>> [pair for pair in conjugations.items()]

>>> [(pair[0][0], pair[1][0][0]) for pair in conjugations.items()]

>>> {pair[0]:pair[1] for pair in conjugations.items()}

>>> def double(x):
    ...   print "double:" , x + x
    ...

>>> print double(2)
???
Types: some definitions and context

• Some historical languages were *untyped*
  – You could, say, divide a string by a number, and the program would continue.
  – The result was still nonsense, of course, and program behavior was completely undefined.
  – This was considered unacceptable

• Modern languages may be *statically typed* or *dynamically typed*
  – “statically typed” means that types are assigned before the program is executed
  – “dynamically typed” means that types are assigned (and type errors caught) at runtime

• Modern languages may be *strongly typed* or *weakly typed*
  – For our purposes, “weakly typed” means the language supports a significant number of implicit type conversions.
    • For example, (5 + “3”) could trigger a conversion from “3” to 3

• For our purposes, Python can be considered
  – strongly typed
  – dynamically typed
def mbar_to_mmHg(pressure):
    return pressure * 0.75006
def abs(x):
    if val < 0:
        return -1 * val
    else:
        return 1 * val
Guess the Types

def debug(x):
    print x
def index(value, somelist):
    i = 0
    for c in somelist:
        if c == value:
            return i
    i = i + 1
Duck Typing

“If it walks like a duck and it talks like a duck, then it must be a duck.”

(Note: this analogy can be misleading!)

At runtime, the operands are checked to make sure they support the requested operation.

>>> 3 + "3"
>>> for i in 5:
...   print i
Takeaway

• Think about types when designing functions, when debugging, when reading code, when writing code....all the time.

• Ask yourself “What operations are being applied to this variable?” and “What values may this variable hold?”
  — A list, or just anything compatible with a for loop?
  — An integer, or anything that can be multiplied by an integer?
Mutable and Immutable Types

```python
>>> def increment(uniquewords, word):
...     """increment the count for word""
...     uniquewords[word] = uniquewords.setdefault(word, 1) + 1

>>> mywords = dict()
>>> increment(mywords, "school")
>>> print(mywords)
{'school': 2}

>>> def increment(value):
...     """increment the value""
...     value = value + 1

>>> myval = 5
>>> increment(myval)
>>> print(myval)
5
```
What’s going on?

Python’s Data Model

• Everything is an object
• Each object has an identity, a type, and a value
  – id(obj) returns the object’s identity
  – type(obj) returns the object’s type
Identity

• The identity of an object can never change
  – (Currently) implemented as the object’s address in memory.
  – You can check to see if two objects are identical with the keyword is
Identity

```python
>>> A = [1]
>>> B = [1]
>>> A == B
True
>>> A is B
False
>>> C = A
>>> A is C
????
>>> A = [1]
>>> B = [1]
>>> A == B
True
>>> A is B
False
```
Type

• The type of an object cannot change
• It specifies two things:
  – what operations are allowed
  – the set of values the object can hold
Back to the Data Model

• Everything is an object
• Each object has an identity, a type, and a value
  – id(obj) returns the object’s identity
  – type(obj) returns the object’s type
• An object’s identity can never change
• An object’s type can never change
• An object’s value can never change, unless it has a mutable type
Example: Tuples vs. Lists

def updaterecord(record, position, value):
    """change the value at the given position""
    record[position] = value

mylist = [1, 2, 3]
mytuple = (1, 2, 3)
updaterecord(mylist, 1, 10)
print(mylist)
updaterecord(mytuple, 1, 10)
print(mytuple)
Why did they do this?

```python
>>> citytuple = ("Atlanta", "GA")
>>> type(citytuple)
<type 'tuple'>
>>> citylist = ["Atlanta", "GA"]
<type 'list'>
>>> weather[citytuple] = "super hot"
>>> weather[citylist] = "super hot"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```
What would this mean?

```python
>>> citylist = ["Atlanta", "GA"]
>>> weather[citylist] = "super hot"
>>> citylist[1] = "Georgia"
>>> weather[["Atlanta", "GA"]]
```
Mutable and Immutable Types

• Immutable
  – numbers, strings, tuples

• Mutable
  – lists and dictionaries

Note: a set is mutable, but a frozenset is immutable
Comprehension Example

```python
names = ["John von Neumann", "Grace Hopper",
        "Alan Turing", "Charles Babbage", "Ada Lovelace"]

split_names = [name.split(" ") for name in names]

last_names = [split_name[1] for split_name in split_names]

last_name_first = [sn[1] + ", " + sn[0] for sn in split_names]
```
Digression: More with Comprehensions

You are given a function

```python
def sim(sequence1, sequence2):
    """Return a number representing the similarity score between the two arguments""
    ...
```

You are given two lists of sequences

```python
org1 = ["ACGTTTCA", "AGGCCTTA", "AAAACCTG"]
org2 = ["AGCTTTGA", "GCCGGAAT", "GCTACTGA"]
```

You want to find all pairs of similar sequences: similarity(A,B) > threshold

```python
[(x,y) for x in org1 for y in org2 if sim(x,y) > threshold]
```
Evaluating Comprehensions

\[(x,y) \text{ for } x \text{ in } org1 \text{ for } y \text{ in } org2 \text{ if } \text{sim}(x,y) > \text{threshold}\]