Strings, I/O and Data Structures

NYC Data Science Academy

OVERVIEW

- String operations
- File Input and Output
- Searching in files
- Data Structures
  - Mutating operations on lists
  - Tuples, sets, and dictionaries
- Expressions vs. statements

Today

- The first class was a kind of “abstract” treatment of lists, the most important data structure in Python.
- Today, we will put our knowledge to work.
  - We can do a lot of ordinary file processing using what we’ve learned.
  - We’ll learn more about data structure in Python.

OVERVIEW

- String operations
- File Input and Output
- Searching in files
- Data Structures
  - Mutating operations on lists
  - Tuples, sets, and dictionaries
- Expressions vs. statements

Review of string basics

- String literals are written using either single quotes ('...') or double quotes ("...").
- Within a string, special symbols can be included by using the escape character (\). Examples are \t for tab and \n for newline. See https://docs.python.org/2.0/ref/strings.html for the complete list.
- The **print** statement produces a more readable output, by omitting the enclosing quotes and printing escaped and special characters:

```
sent = '"Isn\'t," she said.'
sent
'"Isn\'t," she said.'
```

```
print sent
"Isn't," she said.
```

Review of string basics

- If you don't want characters prefaced by \ to be interpreted as special characters, you can use *raw strings* by adding an r before the first quote:

```
print 'C:\some\name'   # here \n means newline!
C:\some
ame
len('C:\some\name')
11
```

```
print r'C:\some\name'  # here \n is two characters
C:\some\name
len(r'C:\some\name')
12
```

Multiline strings

- A string is normally contained on one line, but it can be broken into multiple lines by putting a '\' at the end of each line (but the last):

```
longquote = 'This is a particularly long line \
            that is broken into two lines.'
```

- Python also has multi line quotes: Begin and end the quote with three quotation marks; newlines within the quote are included.

```
longquote = '''This is a particularly long line
            that is broken into two lines.'''
```

Review of string basics

- Strings are similar to lists. You can use subscripts and slicing, and + to join two strings:

```
s = 'my dog has fleas'
s[1]
'y'
s[3:6]
'dog'
s + ', and so do I.'
'my dog has fleas, and so do I.'
```

- Convert a string to a list of single-character strings using the list function:

```
list('dog')
['d', 'o', 'g']
```

Review of string basics

- Going the other way requires a trick (which we'll learn more about today):

```
"".join(_)          # _ refers to the value last printed
'dog'
```

- We have already seen this dot notation when we try to call function after we imported some module. i.e `math.sqrt()`
- Then you might wondering why does an empty string have a function?

Built-in string operations

- Python comes with a large set of functions to perform useful operation on strings. We'll discuss several - the ones we'll need for examples. For a more complete list, see: https://docs.python.org/2/library/string.html.
- First, we need to discuss a new notation that is used for many string operations...

Object method calls

- Python is an *object-oriented* language, which means that many useful types of data - including strings - are *objects*.
  - You can create your own objects. We'll discuss this advanced topic later in this course.
- For the moment the important thing about objects is that they often use a different syntax for function calls than what we've seen before: *dot notation*.
- Dot notation for functions calls on an object is:

```
object.function(arguments, ...)
```

- This is called a method call, but it is really just a function call with a different syntax.

Object method calls

- Since strings are objects, we will often see method calls of the form:

```
string.function(arguments, ...)
```

  - We have already seen one example:

```
"".join(_)
```

    Here, `""` is the object, `join` the function (or "method"), and _ the argument.

- The main thing is *not to get confused by the notation*. A method call is nothing but a function call where the first argument is given - for reasons we can't go into yet - in a different syntax. Just as with any function argument, it can be given as a literal, a variable, or an expression.

Built-in string operations: strip

- **strip()** removes “whitespace” - spaces, tabs, newlines - from the beginning and ending of a string. Note the use of object-oriented syntax:

```
s = '   my dog has fleas   '
s.strip()
'my dog has fleas'
```

Built-in string operations: split

- **split()** is an extremely useful function that splits a string into a list of strings on a delimiter. By default, the delimiter consists of spaces, tabs (\t), and newlines (\n), but this can be given as an argument (this function also uses o-o syntax):

```
s = 'my dog has fleas'
s.split()
['my', 'dog', 'has', 'fleas']

s1 = 'my,dog,has,fleas'
s1.split(',')
['my', 'dog', 'has', 'fleas']
```

Built-in string operations: join

- **join()** concatenates a list of words with intervening occurrences of a delimiter. You can consider join as the inverse of split. For example:

```
l = ['my', 'dog', 'has', 'fleas']

' '.join(l)
'my dog has fleas'

', '.join(l)
'my, dog, has, fleas'
```

- We used join earlier with the empty string as the delimiter:

```
''.join(['d', 'o', 'g'])
'dog'
```

Built-in string operations: replace

- **replace()** returns a string with all occurrences of one string replaced by another. For example:

```
s = 'He is my classmate, and he is learning Python'
s.replace(' is ', ' was ')
'He was my classmate, and he was learning Python'
```

- The object parameter (s) is the string within which the replacement will be done; the first parameter is the string to be replaced by the second parameter.
- If you only want the first occurrence of the string to be replaced, you can add a third argument:

```
s.replace(' is ', ' was ', 1)
'He was my classmate, and he is learning Python'
```

Built-in string operations: find

- **find()** returns the lowest index in string *s* where the substring *sub* is found. Return -1 on failure.

```
s = 'my dog has fleas'
s.find('dog')
3
s.find('dogs')
-1
```

- If we want to find the substring *sub* within a range we need to specify it

```
s.find('a')
8
s.find('a', 9)
14
```

Built-in string operations: % and format

- The % operator is used to create nicely formatted strings from other values. Its syntax is: *format_specifier* % (*tuple of values*).
  - *format_specifier* is a string containing special format symbols, which are used to insert values from the tuple:
    - %s means insert a string
    - %d means insert an integer
    - %f means insert a float
  - The number of format symbols in the format specifier must equal the number of values in the tuple, and each format symbol must match the type of the corresponding value in the tuple.

```
'My name is %s and I am %d years old' % ('Mike', 25)
'My name is Mike and I am 25 years old'
```

Aside: Using method calls in map

- Object notation causes one problem. Suppose we have a method fun on strings, and suppose it has no arguments. So we apply it to string s by writing: s.fun().
- Now suppose we want to apply fun to every element of a list of strings. How can we do that?
  - We can try: `map(fun, L)` but that doesn't work.
- Instead, turn the method into a function:

```
lambda s: s.fun()
```

or

```
def newfun(s):
    return s.fun()
```

Aside: Using method calls in map

- E.g., here is a way to "strip" every string in a list:

```
lis = [" abc  \n", " def", "ghi   "]
map(lambda s: s.strip(), lis)
['abc', 'def', 'ghi']
```

- Here is a way to remove the strings that contain the word "No":

```
# Return true if s does *not* contain 'No'
def has_no_no(s):
   return s.find('No') == -1

L = ['No rain', 'Some snow', 'No sleet']
filter(has_no_no, L)
['Some snow']
```

Built-in string operations: Case conversion

- Python has provided some built-in functions to do case conversions:

```
'ABcd'.lower()      # convert to lowercase
'abcd'

'ABcd'.upper()      # convert to uppercase
'ABCD'

'ABcd'.swapcase()   # swap case
'abCD'

'aCd acD'.title()   # make first letters uppercase
'Acd Acd'
```

Exercise 1: String operations

- Use map to find the first occurrence of the character i in each word in a list:

```
lis = ['today', 'is', 'a', 'nice', 'day']
map( ... , lis)   # fill in the ...
[-1, 0, -1, 1, -1]
```

- Define a function **`find_char`**(*`s, t`*) to find the lowest index of string *t* in *each word* in s:

```
s = 'today is a nice day'
find_char(s, 'i')
[-1, 0, -1, 1, -1]
```

You need to `split` s, and then do as above.

Exercise 1: String operations

- Use map with the formatting operation (%) to turn a list of numbers into a list of strings:

```
lis = [10, 12, 4, 7]
map( ... , lis)   # fill in the ...
['10', '12', '4', '7']
```

- Use filter to find the strings in a list that *do* contain No.

```
L = ['No rain', 'Some snow', 'No sleet']
filter(has_no, L)
['No rain', 'No sleet']
```

OVERVIEW

- String operations
- File Input and Output
- Searching in files
- Data Structures
  - Mutating operations on lists
  - Tuples, sets, and dictionaries
- Expressions vs. statements

Creating files in IPython

- You can create files in IPython:
  - As above, save your notebook and go to the initial iPython screen.
  - In the New menu (upper right), click Text file.
  - Enter your text.
  - Click on “untitled.txt” in the top left to name the file.
  - Select “Save” from the file menu to save it.
  - Click the word Jupyter on top left to return to the iPython screen. You should see your new file listed.
  - Click on “Untitled.ipynb” to return to your notebook.

Exercise: Creating files in IPython

- Follow the instructions above to create file simple.txt, with these lines:

```
I'm line 1,
and I'm line 2.
```

Reading from files

- Reading from files is very simple, because we can treat a file almost as a list of strings.
- To turn a file into a list of strings, simply do this:

```
f = open('simple.txt', 'r')    # 'r' for read
lines = f.readlines()
f.close()
```

- Now that we have the file's contents in a list, we can apply all of our list- and string-processing powers to it. E.g. turn all letters in simple.txt into uppercase:

```
text = ''.join(map(lambda s: s.upper(), lines))
text
"I'M LINE 1,\nAND I'M LINE 2.\n"
```

Reading from files

- Let's take that code apart. simple.txt has two lines:
  - The first three lines read the file into a list, as we've seen. Note that each line still has its ending newline:

```
f = open('simple.txt', 'r')
lines = f.readlines()
f.close()
lines
["I'm line 1,\n", "and I'm line 2."]
```

  - We can apply a function to each line using map. Here we're upper-casing each line:

```
map(lambda s: s.upper(), lines)
["I'M LINE 1,\n", "AND I'M LINE 2."]
```

Reading from files

- We might want to assign the new list to a variable, or maybe back to lines:

```
lines = map(lambda s: s.upper(), lines)
lines
["I'M LINE 1,\n", "AND I'M LINE 2."]
```

because lists are more convenient if we want to do more processing.

- In this case, we just want to get the new text in the form of a string, so we use join:

```
text = ''.join(lines)
text
"I'M LINE 1,\nAND I'M LINE 2."
```

Exercise 2: File input

- The '\n' symbol on the previous slide is quite annoying. Try to get rid of it using the `strip()` function.
- Write a function `e_to_a` to read the contents of a file, and get a list of every line, *with the letter 'e' changed to 'a' in every line*.

```
e_to_a('simple.txt')
["I'm lina 1,", "and I'm lina 2."]
```

Start with the usual code to read the lines of the file, then map `replace` over the lines and return the result.

File output

- Writing output to a file is easy.
  - We open a file for output: `f = open(filename, 'w')`.

    CAUTION: Once this line of code is executed, the file specified would be **ERASED!**
  - Write a string to the file: `f.write(s)`
  - Close the file: `f.close()`

```
f = open('simple.txt', 'w')
f.write('This overwrites the file!')
f.close()
```

File output

- To avoid overwriting the file, we may open a file for appending:

```
f = open('simple.txt', 'a') # 'a' for appending
f.write('\nThis should be the second line.')
f.close()
```

- We often want to read the file first to decide what to be appended to it:

```
f = open('simple.txt', 'r+')
lines = f.readlines()
lines
['This overwrites the file!\n', 'This should be
the second line.']
```

File output

- We may then write a new line into it:

```
f.write('\nThis should be the third line.')
f.close()
```

OVERVIEW

- String operations
- File Input and Output
- Searching in files
- Data Structures
  - Mutating operations on lists
  - Tuples, sets, and dictionaries
- Expressions vs. statements

Searching for words in files

- You may be familiar with the Unix command grep, which is used to search for strings within files. For example:

```
!grep people oldmanandthesea.txt
quite sure no local people would steal from him, the old man thought
He always thought of the sea as _la mar_ which is what people call her
that were as long as the skiff and weighed a ton.  Most people are
him beyond all people.  Beyond all people in the world.  Now we are
table.  There was much betting and people went in and out of the room
many people will he feed, he thought.  But are they worthy to eat him?
did it to keep me alive and feed many people.  But then everything is a
are people who are paid to do it.  Let them think about it.  You were
```

- The txt file is from A Project Gutenberg Canada Ebook.

Searching for words in files

- We can do the same thing in Python, using `filter` and `find`.

```
def grep(word, file):
    f = open(file, 'r')
    lines = f.readlines()
    f.close()
    # has_word is true if line contains word
    has_word = lambda line: line.find(word) != -1
    output = filter(has_word, lines)
    return "".join(output)


print grep('people', 'oldmanandthesea.txt')
```

Exercise 3: Searching in files

- Define `grep2(word1, word2, file)`. It returns the lines that contain both `word1` and `word2`.

OVERVIEW

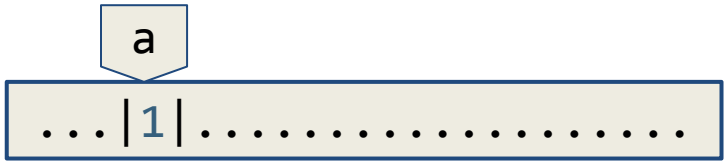
- String operations
- File Input and Output
- Searching in files
- Data Structures
  - Mutating operations on lists
  - Tuples, sets, and dictionaries
- Expressions vs. statements

Data structures

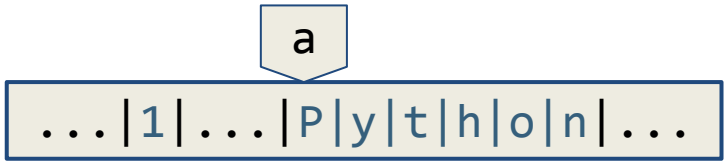
- Lists are the most widely used data structure in Python. But they are not the only one. Other built-in data structures are *sets* and *dictionaries*:
  - Sets - unordered collections without duplicates.
  - Dictionaries - maps from one value (often strings) to another.
- An important feature of Python data structures is that some are *mutable* and some are *immutable*; mutability is a key concept that we will discuss in this section.

Understanding variable assignment

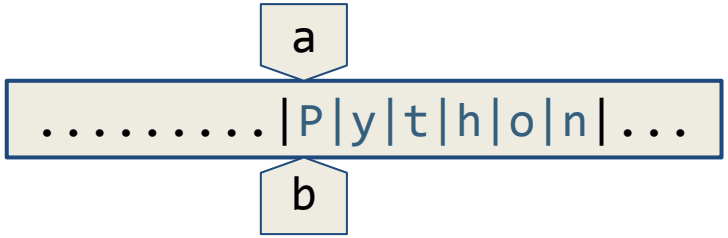
- When you assign a value to a variable, it *tags* the value with the variable name.
  - `a = 1`



- When you re-assign to a variable, it changes the *tag*, *not* the value in memory.
  - `a = 'Python'`



- Assigning one variable to another makes a new tag to the same value.
  - `b = a`



- Thus, if the value in memory changes, both variables will see the change. *But how can the value in memory change?*

Mutability

- Some type of data can be changed, or *mutated*.
  - If an operation mutates a value, you can tell in this way: Assign the value to a variable, then do the operation - *but don't assign its result to the variable*. If the variable has changed, the method mutated it.
- We have not seen any mutating operations on lists. Here are two example of operations not mutating the list `['a', 'b', 'c']`:

```
L = ['a', 'b', 'c']
L[1:]
['b', 'c']
L
['a', 'b', 'c']
```

```
map(lambda s: s.upper(), L)
['A', 'B', 'C']
L
['a', 'b', 'c']
```

Exercise 4: Non-mutating operations

- Try this using list and string operations you have learned:
  - Assign a list or string to a variable (say, L or s).
  - Perform operations on the variable.
  - Note that the variable changes *only* if you re-assign to it.
  - Now assign the variable to another variable:

```
M = L
t = s
```

  - Now perform operations on L and s and assign the result to L or s, e.g. "`L = L[2:]`" or "`s = s.upper()`". *Do M or t change?*

OVERVIEW

- String operations
- File Input and Output
- Searching in files
- Data Structures
  - Mutating operations on lists
  - Tuples, sets, and dictionaries
- Expressions vs. statements

Mutating operations on lists

- Lists are a mutable data type - we just haven't introduced any of the mutating operations. We now introduce several of them.

Element-wise assignment

- You can change an individual element of a list by assignment:

```
skills = ['Python','SAS','Hadoop']
skills[1] = 'R'     # no assignment to skill itself
skills
['Python', 'R', 'Hadoop']
```

- You can also observe that a mutation has happened by assigning to a different variable and viewing that variable after the mutating operation:

```
skills = ['Python','SAS','Hadoop']
my_skills = skills
skills[1] = 'R'     # no assignment to my_skills
my_skills
['Python', 'R', 'Hadoop']
```

List extension

- We have always added to a list by using +, which is non-mutating:

```
L = ['a', 'b', 'c']
M = L + ['d']       # assignment to M doesn't change L
L
['a', 'b', 'c']
L = L + ['d']       # only assignment to L can change L
L
['a', 'b', 'c', 'd']
```

- The append operation mutates a list:

```
L.append('e')       # no assignment to L
L                   # yet L has changed
['a', 'b', 'c', 'd', 'e']
```

Other mutating operations on lists

- `lis.remove(x)` removes the first item from the list whose value is *x*.

```
L
['a', 'b', 'c', 'd', 'e']
L.remove('c')
L
['a', 'b', 'd', 'e']
```

- `del L[i]` removes item *i* from L.

```
del L[0]
L
['b', 'b', 'e']
```

Other mutating operations on lists

- `lis.insert(`*i*`, `*x*`)` inserts *x* so that it is at location *i* in the list. (If *i* is out of bounds, it inserts it in the closest place it can.)

```
L = ['a', 'b', 'c']
L.insert(2, 'd')
L
['a', 'b', 'd', 'c']
L.insert(10, 'e')
L
['a', 'b', 'd', 'c', 'e']
L.insert(-10, 'f')
L
['f', 'a', 'b', 'd', 'c', 'e']
```

Other mutating operations on lists

- We have already seen `sorted(`*lis*`)`, which is a non-mutating sort operation:

```
lis = [4, 2, 6, 1]
sorted(lis)
[1, 2, 4, 6]
lis
[4, 2, 6, 1]
```

- *lis*`.sort()` is a mutating sort operation:

```
lis.sort()
lis
[1, 2, 4, 6]
```

Side effects of mutating operations

- It follows from what we've seen that a mutating operation applied to a list L can change the value of another variable if that variable is pointing to the same memory location as L:

```
lis = [4, 2, 6, 1]
lis2 = lis
lis.sort()
lis2
[1, 2, 4, 6]
```

`lis2` would not have changed if we had written "`lis = sorted(lis)`".

- This is called a *side effect* of the mutating operation. Programmers try to avoid side effects, because it is difficult to understand code when variables can change without even being mentioned.

Values of mutating operations

- Note that the mutating operations we have seen have no value, or rather, their value is None. Try:

```
print lis.sort()
print lis.append(4)
```

- It follows that we cannot use mutating operations in a map or filter, because those depend upon the value of the expression. This is an attempt to extend every element of a nested list:

```
L = [[1], [2], [3]]
map(lambda l: l.append(4), L)
[None, None, None]
```

- Later in this class, we will see how to apply mutating operations to every element of a list.

Aside: Customizing sort order

- `sort` and `sorted` use the first element as the primary sort key, the second element as the second sort key, etc., and they sort in ascending order. You can customize the sort using two different arguments:
  - Sort on a user-defined key:

```
staff =[['Lucy','A',9], ['John','B',3], ['Peter','A',6]]
sorted(staff, key = lambda x: x[2])  # key is ID number
[('John', 'B', 3), ('Peter', 'A', 6),('Lucy', 'A', 9)]
```

User-defined functions that mutate data

- You can define functions that use mutating operations. If the purpose of a function is to perform a mutating operation, it does not need a return value.
- This function sorts a nested list, using the given element of each sublist as the sort key:

```
def sort_on_field(lis, fld):
    lis.sort(key = lambda x: x[fld])

L = [['a', 4], ['b', 1], ['c', 7], ['d', 3]]
sort_on_field(L, 1)
L
[['b', 1], ['d', 3], ['a', 4], ['c', 7]]
```

- It has no `return`, and does not produce a value.

Exercise 5: Mutating functions

- Write a function to switch the i^{th} and j^{th} items in a list.

```
def switch_item(L, i, j):
    # your code goes here

my_list = ['first', 'second', 'third', 'fourth']
switch_item(my_list, 1, -1)
my_list
['first', 'fourth', 'third', 'second']
```

OVERVIEW

- String operations
- File Input and Output
- Searching in files
- Data Structures
  - Mutating operations on lists
  - Tuples, sets, and dictionaries
- Expressions vs. statements

Python's data types

- We can now explain the other data types of Python.
  - **Tuples**: Tuples are like lists, but are *immutable*.
  - **Sets**: Also like lists, except that they do not have duplicate elements. *Immutable*.
  - **Dictionaries**: These are tables that associate values with keys (usually strings). *Mutable*.
  - **Strings**: Like lists of characters. *Immutable*.

Strings are immutable

- This is really all we want to say about strings.

```
company = 'NYC DataScience Academy'
company[0] = 'A'

TypeError: 'str' object does not support item assignment
```

Tuples

- Tuples are similar to lists, but they are immutable.
- Tuples are written with parentheses instead of square brackets.

```
courses = ('Programming', 'Stats', 'Math')
courses[2] = 'Algorithms'

TypeError: 'tuple' object does not support item assignment
```

- Tuples support all the non-mutating list operations:

```
courses[1:]
('Stats', 'Math')
map(lambda s: s.upper(), courses)
['PROGRAMMING', 'STATS', 'MATH']
```

Aside: Multiple assignment

- Tuples and lists both allow a shorthand for assignment that allows all the elements of the tuple or list to be assigned to variables at once:

```
(a,b) = (1,2)   # works with lists also
a
1
b
2
```

- This provides a handy way to swap variables:

```
(a,b) = (b,a)
a
2
b
1
```

Set

- A set is an *unordered* collection with no duplicate elements. Sets are immutable.
- To create a set, you can use either curly braces or the `set()` function.

```
vowels = {'u','a','e','i','o','u','i'}
vowels
set(['i', 'e', 'u', 'a', 'o'])
fruit = set(['apple', 'orange', 'apple', 'pear'])
fruit
set(['orange', 'pear', 'apple'])
```

Set operations

- Sets support non-mutating list operations, as long as they don't depend on order:

```
primes = {2, 3, 5, 7}

primes[2]
TypeError: 'set' object does not support indexing

sum(primes)
17

map(lambda x: x*x, primes)
[4, 9, 25, 49]      # order is not guaranteed
```

Set operations

- Sets have mathematical operations like union (|), intersection (&), difference (-), and symmetric difference (^).

```
set_1 = {'a', 'b', 'c'}
set_2 = {'b', 'c', 'd'}

set_1 | set_2        # union
set(['a', 'c', 'b', 'd'])
set_1 & set_2        # intersection
set(['c', 'b'])
set_1 - set_2        # difference
set(['a'])
set_1 ^ set_2        # symmetric difference (a-b | b-a)
set(['a', 'd'])
```

Dictionaries

- A dictionary is a set of keys with associated values. Each key can have just one value associated with it. Dictionaries are mutable.
  - Any immutable object can be a key, including numbers, strings, and tuples of numbers or strings. Strings are most common.
  - Any object can be a value.
- Dictionaries are written in set braces (like sets), with the key/value pairs separated by colons:

```
employee = {'sex': 'male', 'height': 6.1, 'age': 30}
```

- The most important operation on dictionaries is key lookup:

```
employee['age']
30
```

Dictionaries

- We can add new *key: value* pairs to the dictionary:

```
employee['city'] = 'New York'
employee
{'city': 'New York', 'age': 30, 'height': 6.1, \
'sex': 'male'}
```

- It is illegal to access a key that is not present:

```
employee['weight']
KeyError: 'weight'
```

but you can check if a key is present using the `in` operator:

```
'weight' in employee
False
```

Dictionaries

- For convenience, you can construct a dictionary from a list of tuples:

```
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'guido': 4127, 'jack': 4098, 'sape': 4139}
```

- You can also get a list of the keys, the values, or all key/value pairs:

```
employee = {'sex': 'male', 'height': 6.1, 'age': 30}
employee.keys()
['age', 'height', 'sex']
employee.values()
[30, 6.1, 'male']
employee.items()
[('age', 30), ('height', 6.1), ('sex', 'male')]
```

Why dictionaries?

- You could represent a table as a list of pairs, use append to add items, and use filter to look them up:

```
employee = [('sex', 'male'), ('height', 6.1), ('age', 30)]
filter(lambda x: x[0]=='sex', employee)[0][1]
'male'
```

- The big advantage of dictionaries is that they are extremely efficient - much more efficient than using filter.

Exercise 6: Tuples and Dictionaries

- Given the following dictionary:

```
inventory = {'pumpkin' : 20, 'fruit' : ['apple', 'pear'],
             'vegetable' : ['potato','onion','lettuce']}
```

- Modify `inventory` as follows:
  - Add a `meat` inventory item containing 'beef', 'chicken', and 'pork'.
  - Sort the vegetables (Recall the `sorted` function.)
  - Add five more pumpkins.
- After these changes, `inventory` is:

```
{'vegetable': ['lettuce', 'onion', 'potato'],
'fruit': ['apple', 'pear'], 'meat': ['beef',
'chicken', 'pork'], 'pumpkin': 25}
```

OVERVIEW

- String operations
- File Input and Output
- Searching in files
- Data Structures
  - Mutating operations on lists
  - Tuples, sets, and dictionaries
- Expressions vs. statements

Expressions vs. statements

- A key property of virtually all programming languages is that they have two basic categories of syntactic constructs:
  - *Expressions* produce values. This includes ordinary expressions like `3+4`, as well as more complex expressions like `map(lambda x: x+1, [3, 5, 7])`. In both cases, we can ask "what is the value of that expression?" and get a sensible answer (7 and [4, 6, 8], resp.).
  - *Statements* cause an action - a "side effect" - to happen. We have seen two statements: assignments and `print`.
- In the programming style we have adopted up to now, we have used mostly expressions, and have used assignments and `print` only rarely.

def vs. lambda

- The biggest difference between def-style and lambda-style function definitions is this:
  - *The bodies of def-style function definitions are statements, while the bodies of lambda-style functions are expressions.*
- lambda functions are limited because their bodies cannot contain statements, which means they cannot have assignments to variables. A function that is complicated enough will have assignments, so long function definitions are nearly always written in def style.

Summary: statements vs. expressions

- The difference between statements and expressions is fundamental
  - Expressions return values.
  - Statements perform actions.
- Most often, the action performed by a statement is to assign new values to variables, or change the values of variables (by invoking mutating operations).