A Dictionary Example

Problem: count the frequency of each word in text read from the standard input, print results

- Three versions of increasing complexity
- wf1.py is a simple start
- wf2.py uses a common idiom for default values
- wf3.py sorts the output alphabetically

Dictionary example: wf1.py

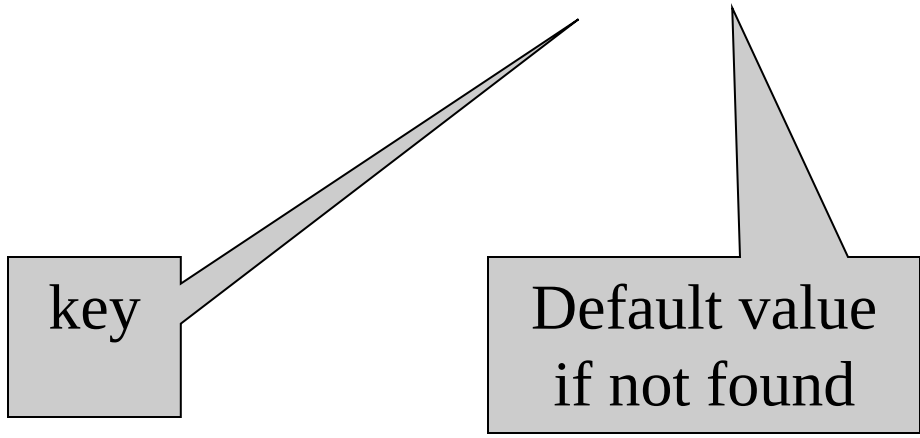
```
import sys
freq = {}    # frequency of words in text
for word in input().split(" "):
        if word in freq:
            freq[word] = 1 + freq[word]
        else:
            freq[word] = 1
print (freq)
```

Dictionary example wf1.py

```
import sys
freq = {}    # frequency of words in text
for word in input().split(" "):
        if word in freq:
            freq[word] = 1 + freq[word]
        else:
            freq[word] = 1
print (freq)
```

Dictionary example wf2.py

```
import sys
freq = {}    # frequency of words in text
for word in input().split(" "):
        freq[word] = 1 + freq.get(word,0)
print (freq)
```



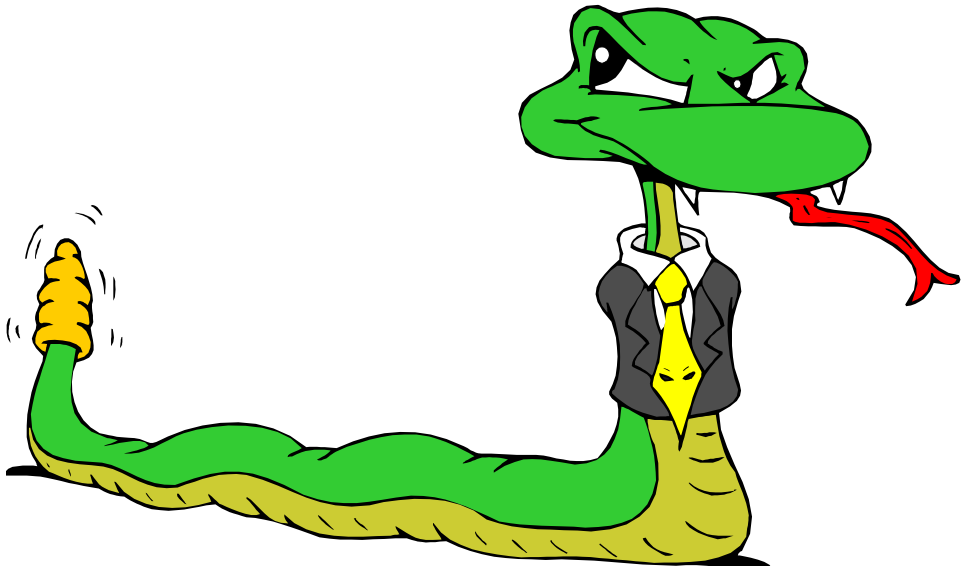
Dictionary example wf3.py

```
import sys
freq = {}    # frequency of words in text

for word in input().split(" "):
     freq[word] = 1 + freq.get(word,0)

for w in sorted(freq.keys()):
    print (w, freq[w])
```

Functions in Python



Defining Functions

Function definition begins with “def.”

Function name and its arguments.

```
def get_final_answer(filename):
    """Documentation String"""
    line1
    line2
    return total_counter
```

Colon.

The indentation matters…
First line with less indentation is considered to be outside of the function definition.

The keyword ‘return’ indicates the value to be sent back to the caller.

No header file or declaration of types of function or arguments

Python and Types

- Dynamic typing*:* Python determines the data types of *variable bindings* in a program automatically
- Strong typing: But Python's not casual about types, it enforces the types of *objects*
- For example, you can't just append an integer to a string, but must first convert it to a string

```
x = "the answer is "  # x bound to a string
y = 23                # y bound to an integer.
print (x + y)         # Python will complain!
```

Calling a Function

- The syntax for a function call is:

```
>>> def myfun(x, y):
        return x * y
>>> myfun(3, 4)
12
```

Functions without returns

- *All* functions in Python have a return value, even if no *return* line inside the code
- Functions without a *return* return the special value *None*
  - *None* is a special constant in the language
  - *None* is used like *NULL*, *void*, or *nil* in other languages
  - *None* is also logically equivalent to False
  - The interpreter doesn't print *None*

Function overloading? No.

- There is no function overloading in Python
  - Unlike Java, a Python function is specified by its name alone
  - Two different functions can’t have the same name, even if they have different arguments

Default Values for Arguments

- You can provide default values for a function's arguments
- These arguments are optional when the function is called

```
>>> def myfun(b, c=3, d="hello"):
            return b + c
>>> myfun(5,3,"hello")
>>> myfun(5,3)
>>> myfun(5)
```

All of the above function calls return 8

Keyword Arguments

- You can call a function with some or all of its arguments out of order as long as you specify their names

```
>>> def myfun(a, b, c):
        return a-b
>>> myfun(2, 1, 43)
  1
>>> myfun(c=43, b=1, a=2)
  1
>>> myfun(2, c=43, b=1)
  1
```

Functions are first-class objects

Functions can be used as any other datatype, eg:

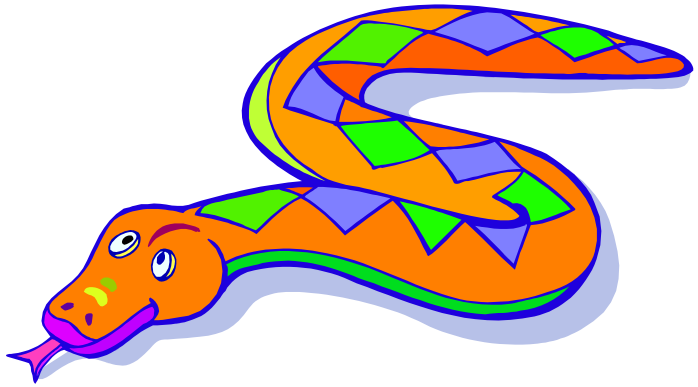
- Arguments to function
- Return values of functions
- Assigned to variables
- Parts of tuples, lists, etc

```
>>> def square(x):
        return x*x

>>> def applier(q, x):
        return q(x)

>>> applier(square, 7)
49
```

Logical Expressions



True and False

- *True* and *False* are constants in Python.
- Other values equivalent to *True* and *False*:
  - *False*: zero, *None*, empty container or object
  - *True*: non-zero numbers, non-empty objects
- Comparison operators: ==, !=, <, <=, etc.
  - X and Y have same value: `X == Y`
  - Compare with `X is Y` :
    - — X and Y are two variables that refer to the *identical same object.*

Boolean Logic Expressions

- You can also combine Boolean expressions.
  - *True* if a is True and b is True: `a and b`
  - *True* if a is True or b is True: `a or b`
  - *True* if a is False: `not a`
- Use parentheses as needed to disambiguate complex Boolean expressions.

Conditional Expressions

```
x = true_value if condition else false_value
```

- Uses lazy evaluation:
  - First, `condition` is evaluated
  - If *True*, `true_value` is evaluated and returned
  - If *False*, `false_value` is evaluated and returned
- Standard use:

```
x = (true_value if condition else false_value)
```

Control of Flow

if Statements

```
if x == 3:
    print ("X equals 3.")
elif x == 2:
    print ("X equals 2.")
else:
    print ("X equals something else.")
print ("This is outside the 'if'.")
```

Be careful! The keyword *if* is also used in the syntax of filtered *list comprehensions*. Note:

- Use of indentation for blocks
- Colon (*:*) after boolean expression

while Loops

```
>>> x = 3
>>> while (x < 5):
        print (x, "still in the loop”)
        x = x + 1
3 still in the loop
4 still in the loop
>>> x = 6
>>> while (x < 5):
        print (x, "still in the loop”)


>>>
```

break and *continue*

- You can use the keyword *break* inside a loop to leave the *while* loop entirely.

- You can use the keyword *continue* inside a loop to stop processing the current iteration of the loop and to immediately go on to the next one.

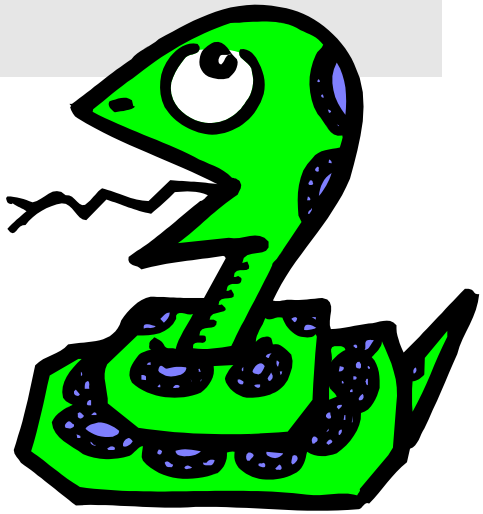
assert

- An *assert* statement will check to make sure that something is true during the course of a program.
  - If the condition if false, the program stops

    (more accurately: the program throws an exception)

```
assert(number_of_players < 5)
```

For Loops



For Loops 1

- A for-loop steps through each of the items in a collection type, or any other type of object which is "iterable"

```
for <item> in <collection>:
  <statements>
```

- If <collection> is a list or a tuple, then the loop steps through each element of the sequence
- If <collection> is a string, then the loop steps through each character of the string

```
for someChar in "Hello World":
    print someChar
```

For Loops 2

```
for <item> in <collection>:
  <statements>
```

- <item> can be more than a single variable name
- When the <collection> elements are themselves sequences, then <item> can match the structure of the elements.
- This multiple assignment can make it easier to access the individual parts of each element

```
for (x,y) in [(a,1),(b,2),(c,3),(d,4)]:
  print x
```

For loops & the *range()* function

- Since a variable often ranges over some sequence of numbers, the *range()* function returns a list of numbers from 0 up to but not including the number we pass to it.
- list(range(5)) returns [0,1,2,3,4]
- So we could say:

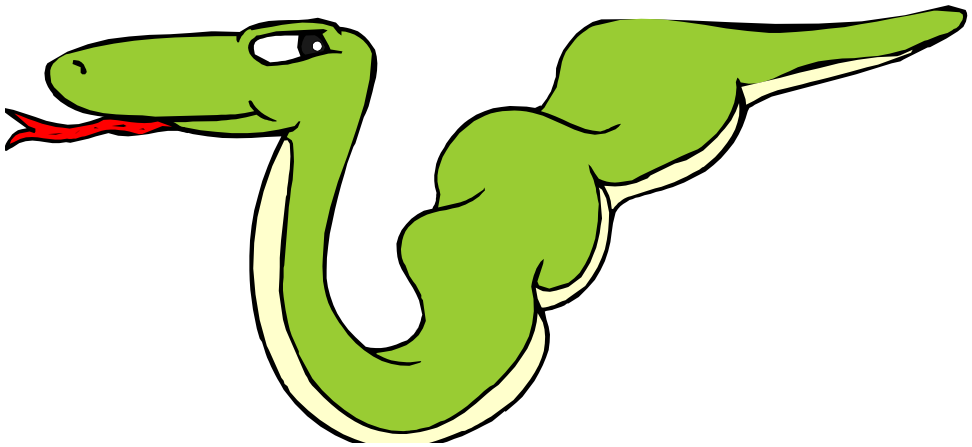
```
for x in range(5):
    print x
```

- (There are more complex forms of *range()* that provide richer functionality…)

For Loops and Dictionaries

```
>>> ages = { "Sam" : 4, "Mary" : 3, "Bill" : 2 }
>>> ages
{'Bill': 2, 'Mary': 3, 'Sam': 4}
>>> for name in ages.keys():
        print (name, ages[name])
Bill 2
Mary 3
Sam 4
>>>
```

Assignment and Containers



Multiple Assignment with Sequences

- We've seen multiple assignment before:

```
>>> x, y = 2, 3
```

- But you can also do it with sequences.
- The type and "shape" just has to match.

```
>>> (x, y, (w, z)) = (2, 3, (4, 5))
>>> [x, y] = [4, 5]
```

Empty Containers 1

- Assignment creates a name, if it didn't exist already.

  `x = 3` Creates name x of type integer.

- Assignment is also what creates named references to containers.

```
>>> d = {'a':3, 'b':4}
```

- We can also create empty containers:

```
>>> li = []
>>> tu = ()
>>> di = {}
```

Note: an empty container is *logically* equivalent to False. (Just like None.)

- These three are empty, but of different *types*

Empty Containers 2

Why create a named reference to empty container?

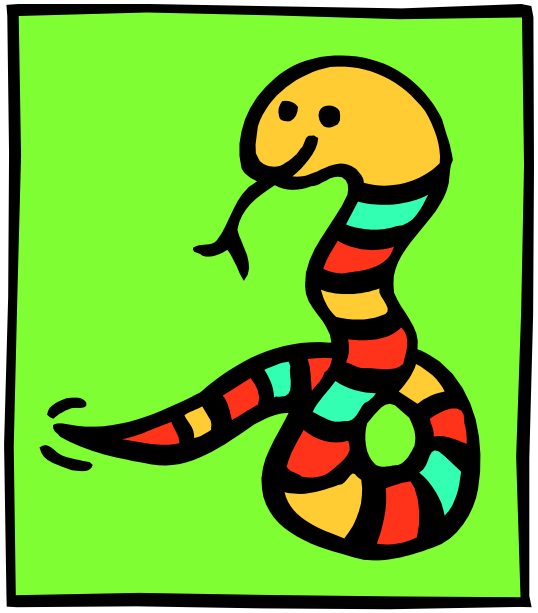
- To initialize an empty list, e.g., before using append
- This would cause an unknown name error if a named reference to the right data type wasn't created first

```
>>> g.append(3)
```

Python complains here about the unknown name 'g'!

```
>>> g = []
>>> g.append(3)
>>> g
  [3]
```

String Operations



String Operations

- A number of methods for the string class perform useful formatting operations:

```
>>> "hello".upper()
'HELLO'
```

- Check the Python documentation for many other handy string operations.

- Helpful hint: use `<string>.strip()` to strip off final newlines from lines read from files

String Formatting Operator: %

- The operator *%* allows strings to be built out of many data items a la "fill in the blanks"
  - Allows control of how the final output appears
  - For example, we could force a number to display with a specific number of digits after the decimal point
- Very similar to the sprintf command of C.

```
>>> x = "abc"
>>> y = 34
>>> "%s xyz %d" % (x, y)
'abc xyz 34'
```

- The tuple following the *%* operator used to fill in blanks in original string marked with *%s* or *%d*.
- Check Python documentation for codes

Printing with Python

- You can print a string to the screen using *print*
- Using the % operator in combination with print, we can format our output text

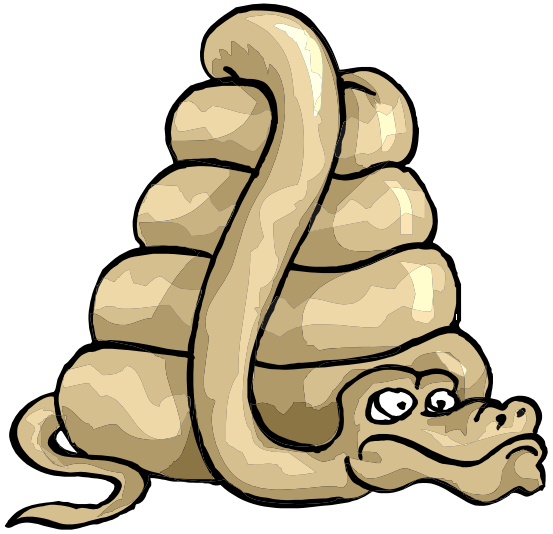
```
>>> print  ("%s xyz %d"  %  ("abc", 34))
abc xyz 34
```

- *Print* adds a newline to the end of the string. If you include a list of strings, it will concatenate them with a space between them

```
>>> print ("abc")
abc
```

```
>>> print ("abc", "def")
abc def
```

String Conversions



Join and Split

- Join turns a list of strings into one string

  <separator_string>.join(<some_list>)

  ```
  >>> ";".join( ["abc", "def", "ghi"] )
  "abc;def;ghi"
  ```

- Split turns one string into a list of strings

  <some_string>.split(<separator_string>)

  ```
  >>> "abc;def;ghi".split( ";" )
  ["abc", "def", "ghi"]
  ```

- Note the inversion in the syntax

Convert Anything to a String

- The builtin str() function can convert an instance of any data type into a string.
- You define how this function behaves for user-created data types
- You can also redefine the behavior of this function for many types.

```
>>> “Hello ” + str(2)
“Hello 2”
```

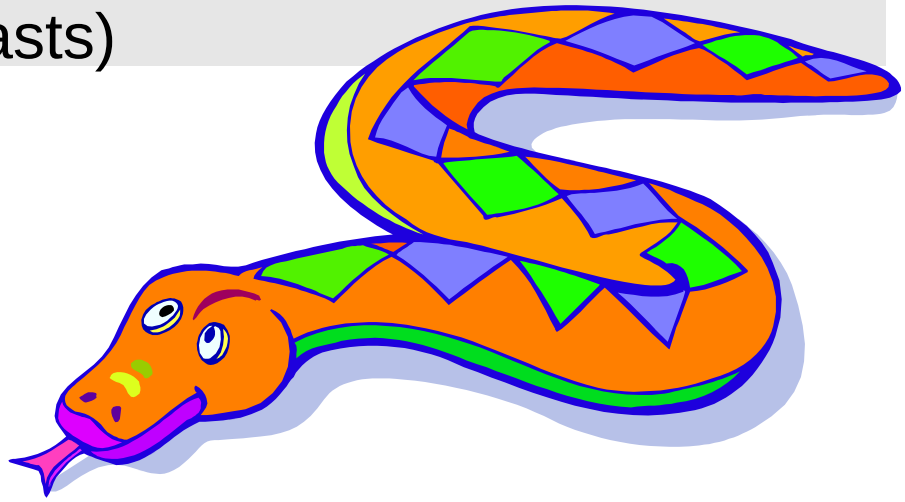
Exercise for home

- Write a function that returns the text from the morse code and vice-versa.
- http://en.wikipedia.org/wiki/Morse_code

- http://www.learnpython.org

List Comprehensions

(on your own, for enthusiasts)



Python's higher-order functions

- Python supports higher-order functions that operate on lists

```
>>> def square(x):
        return x*x
>>> def even(x):
        return 0 == x % 2
>>> list(map(square, range(10,20)))
[100, 121, 144, 169, 196, 225, 256, 289, 324, 361]
>>> list(filter(even, range(10,20)))
[10, 12, 14, 16, 18]
>>> list(map(square, filter(even, range(10,20))))
[100, 144, 196, 256, 324]
```

- But many Python programmers prefer to use list comprehensions, instead

List Comprehensions

- A *list comprehension* is a programming language construct for creating a list based on existing lists
  - Haskell, Erlang, Scala and Python have them
- Why "comprehension"? The term is borrowed from math's *set comprehension* notation for defining sets in terms of other sets
- A powerful and popular feature in Python
  - Generate a new list by applying a function to every member of an original list
- Python's notation:
  [expression for name in list]

List Comprehensions

- The syntax of a *list comprehension* is somewhat tricky

```
[x-10 for x in grades if x>0]
```

- Syntax suggests that of a *for*-loop, an *in* operation, or an *if* statement
- All three of these keywords ('*for*' , '*in*' , and '*if*') are also used in the syntax of forms of list comprehensions

[expression for name in list]

List Comprehensions

```
>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]
[6, 12, 4, 14]
```

Note: Non-standard colors on next few slides clarify the list comprehension syntax.

[expression `for` name `in` list]

- Where expression is some calculation or operation acting upon the variable name.
- For each member of the list, the list comprehension
  1. sets name equal to that member,
  2. calculates a new value using expression,
- It then collects these new values into a list which is the return value of the list comprehension.

[expression for name in list]

List Comprehensions

- If list contains elements of different types, then expression must operate correctly on the types of all of list members.
- If the elements of list are other containers, then the name can consist of a container of names that match the type and “shape” of the list members.

```
>>> li = [(‘a’, 1), (‘b’, 2), (‘c’, 7)]
>>> [ n * 3 for (x, n) in li]
[3, 6, 21]
```

[expression for name in list]

List Comprehensions

- expression can also contain user-defined functions.

```
>>> def subtract(a, b):
      return a - b

>>> oplist = [(6, 3), (1, 7), (5, 5)]
>>> [subtract(y, x) for (x, y) in oplist]
[-3, 6, 0]
```

[expression for name in list]

Syntactic sugar

List comprehensions can be viewed as syntactic sugar for a typical higher-order functions

`[` expression `for` name `in` list `]`

list(map(lambda name: expression, list))

`[` 2*x+1 `for` x `in` [10, 20, 30] `]`

list(map(lambda x: 2*x+1, [10, 20, 30]))

Filtered List Comprehension

- Filter determines whether expression is performed on each member of the list.

- For each element of list, checks if it satisfies the filter condition.

- If the filter condition returns *False*, that element is omitted from the list before the list comprehension is evaluated.

[expression for name in list if filter]

Filtered List Comprehension

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem*2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition
- So, only 12, 14, and 18 are produce.

[expression for name in list if filter]

More syntactic sugar

Including an if clause begins to show the benefits of the sweetened form

[expression for name in list if filt]

list(map(lambda name . expression, filter(filt, list)))

[2*x+1 for x in [10, 20, 30] if x > 0]

list(map(lambda x: 2*x+1,
filter(lambda x: x > 0 , [10, 20, 30])))

Nested List Comprehensions

- Since list comprehensions take a list as input and produce a list as output, they are easily nested

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
        [item+1 for item in li] ]
[8, 6, 10, 4]
```

- The inner comprehension produces: [4, 3, 5, 2]
- So, the outer one produces: [8, 6, 10, 4]

[expression for name in list]

Syntactic sugar

```
[ e1 for n1 in [ e1 for n1 list  ] ]
list(map( lambda n1: e1,
      map( lambda n2: e2, list ) ) )

[2*x+1 for x in [y*y for y  in [10, 20, 30]]]
list(map( lambda x: 2*x+1,
      map( lambda y: y*y, [10, 20, 30] )))
```

Split & Join with List Comprehensions

Split and join can be used in a list comprehension in the following Python idiom:

```
>>> " ".join( [s.capitalize() for s in "this is a test ".split( )] )
'This Is A Test '
>>> # For clarification:
>>> "this is a test" .split( )
['this', 'is', 'a', 'test']
>>> [s.capitalize() for s in "this is a test" .split()]
['This', 'Is', 'A', 'Test' ]
```