Full Python Tutorial

- **Developed by Guido van Rossum in the early 1990s**
  - In July 2018, Van Rossum stepped down as the leader in the language community after 30 years.
- **Named after Monty Python**
- **Available for download from http://www.python.org**

Python

- Basic and Sequence Datatypes
- Dictionaries
- Control Structures
- List Comprehension
- Function definition
- Positional and keyword arguments of functions
- Functional Programming in Python
- Iterators and Generators
- Using higher order functions: Decorators
- Classes and Instances

Language features

- **Indentation instead of braces**
- **Several sequence types**
  - Strings ′ … ′ : made of characters, immutable
  - Lists […] : made of anything, mutable
  - Tuples (…) : made of anything, immutable
- **Powerful subscripting (slicing)**
- **Functions are independent entities (not all functions are methods)**
- **Exceptions as in Java**
- **Simple object system**
- **Iterators (like Java) and *generators***

Why Python?

- **Good example of *scripting language***
- **“Pythonic” style is very concise**
- **Powerful but unobtrusive object system**
  - *Every* value is an object
- **Powerful collection and iteration abstractions**
  - Dynamic typing makes generics easy

Dynamic typing – the key difference

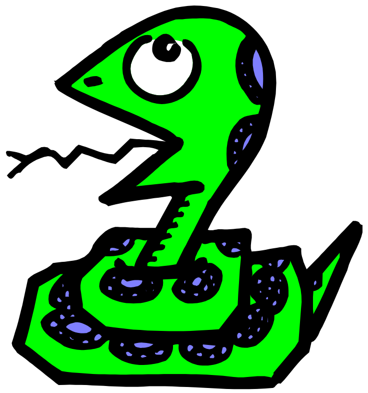
- **Java: *statically typed***
  - Variables are declared to refer to objects of a given type
  - Methods use type signatures to enforce contracts
- **Python**
  - Variables come into existence when first assigned to
  - A variable can refer to an object of any type
  - All types are (almost) treated the same way
  - *Main drawback*: type errors are only caught at runtime

Recommended Reading

- **On-line Python tutorials**
  - The Python Tutorial (http://docs.python.org/tutorial/)
    - Dense but more complete overview of the most important parts of the language
    - See course home page for others
- **PEP 8- Style Guide for Python Code**
  - http://www.python.org/dev/peps/pep-0008/
  - The official style guide to Python, contains many helpful programming tips
- ***Many* other books and on-line materials**
  - If you have a specific question, try Google first

Technical Issues

Installing & Running Python



Which Python?

- **Python 2.7**
  - Last stable release before version 3
  - Implements some of the new features in version 3, but fully backwards compatible
- **Python 3**
  - Released in December 2008
  - Many changes (including incompatible changes)
  - *Much* cleaner language in many ways
  - Strings use Unicode, not ASCII
  - But: A few important third party libraries are not yet compatible with Python 3 right now
  - "End-of-life" date of Python 2 set initially at 2015, then postponed to 2020

The Python Interpreter

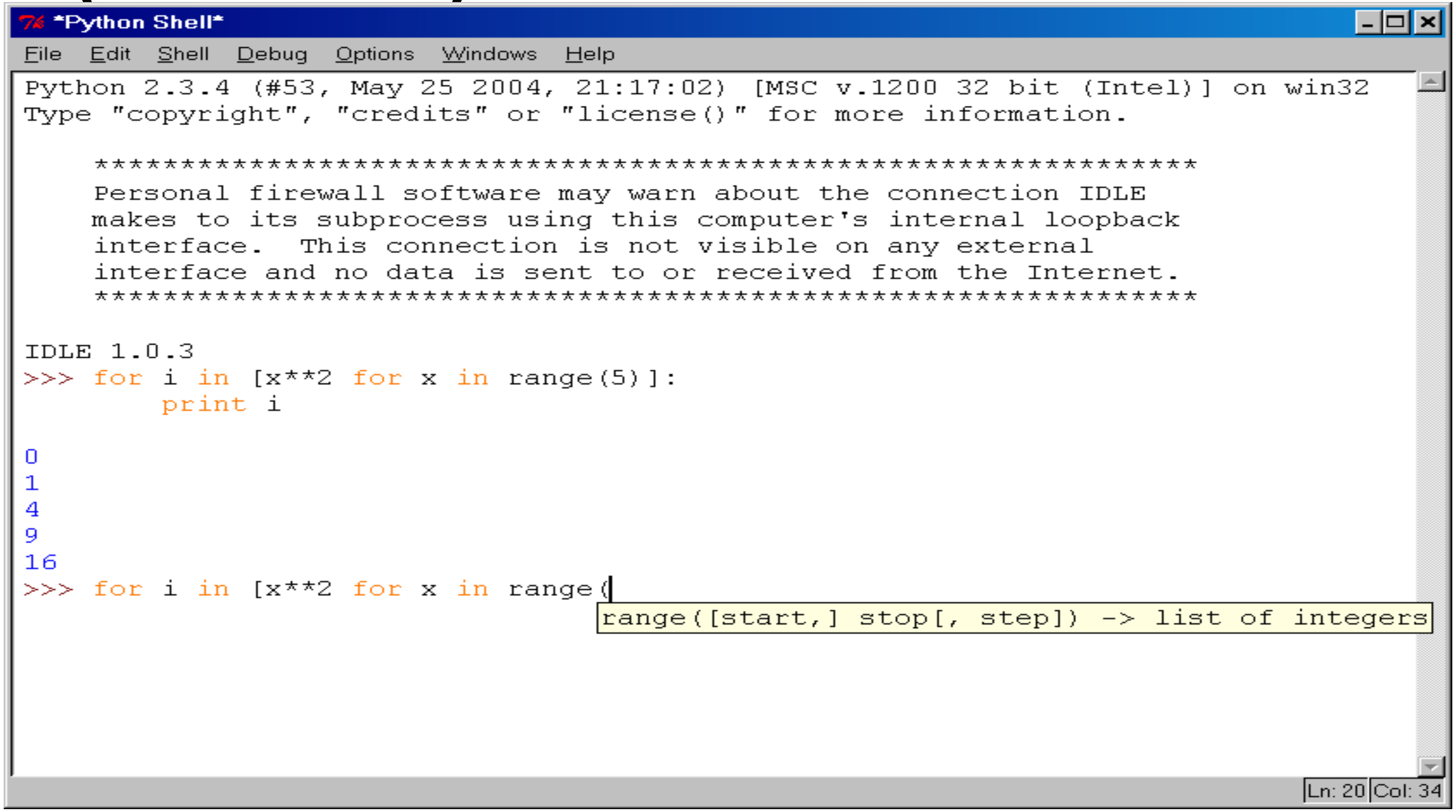
- **Interactive interface to Python**

```
% python
  Python 2.5 (r25:51908, May 25 2007, 16:14:04)
  [GCC 4.1.2 20061115 (prerelease) (SUSE Linux)] on linux2
  Type "help", "copyright", "credits" or "license" for more information.
  >>>
```

- **Python interpreter evaluates inputs:**

```
>>> 3*(7+2)
27
```

The IDLE GUI Environment (Windows)



```
Python 2.3.4 (#53, May 25 2004, 21:17:02) [MSC v.1200 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.0.3
>>> for i in [x**2 for x in range(5)]:
        print i


0
1
4
9
16
>>> for i in [x**2 for x in range(
                                  range([start,] stop[, step]) -> list of integers
```

IDLE Development Environment

- Shell for interactive evaluation.
- Text editor with color-coding and smart indenting for creating Python files.
- Menu commands for changing system settings and running files.

Running Programs on UNIX

% `python filename.py`

You can create python files using emacs.

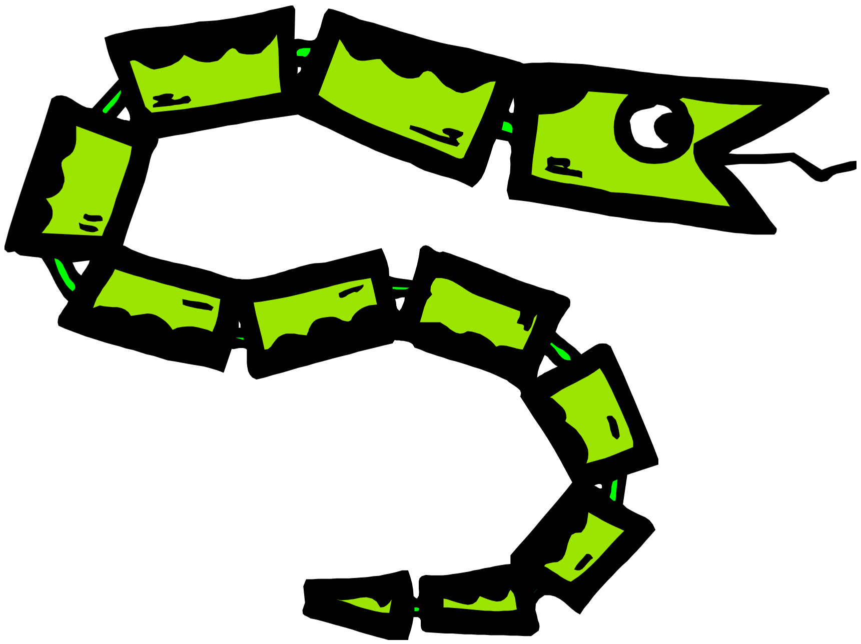
(There's a special Python editing mode.

`M-x python-mode`)

To make a python file executable, make this text the first line of the file :

#!/usr/bin/python

The Basics



A Code Sample (in IDLE)

```
x = 34 - 23                 # A comment.
y = "Hello"                 # Another one.
z = 3.45
if z == 3.45 or y == "Hello":
     x = x + 1
     y = y + " World"    # String concat.

print(x)                 # [Py2] also  print x, no brackets
print(y)
```

Enough to Understand the Code

- **Indentation matters to the meaning of the code:**
  - Block structure indicated by indentation
- **The first assignment to a variable creates it.**
  - Variable types don't need to be declared.
  - Python figures out the variable types on its own.
- **Assignment uses `=` and comparison uses `==`.**
- **For numbers `+ - * / %` are as expected.**
  - Special use of `+` for string concatenation.
  - Special use of `%` for string formatting (as with **printf** in C)
- **Logical operators are words (`and, or, not`)**
  ***not* symbols**
- **Simple printing can be done with `print()`.**

Basic Datatypes

- Integers (default for numbers)

```
z = 5 // 2     # Answer is 2, integer division.
```

- Floats

```
x = 3.456
k = 5 / 2   # k = 2.5 in [Py3], k = 2 in [Py2]
```

- Strings
  - Can use "" or ' ' to specify.
    `"abc"` `'abc'` (Same thing.)
  - Unmatched can occur within the string.
    `"matt's"`
  - Use triple double-quotes for multi-line strings or strings than contain both ' and " inside of them:
    `"""a'b"c"""`

Whitespace

Whitespace is meaningful in Python: especially indentation and placement of newlines.

- ***Use a newline to end a line of code.***
  - Use `\` when must go to next line prematurely.
- **No braces `{ }` to mark blocks of code in Python… *Use consistent indentation instead.***
  - *The first line with less indentation is outside of the block.*
  - *The first line with more indentation starts a nested block*
- **Often a colon “`:`” appears at the start of a new block. (E.g. for function and class definitions.)**

Comments

- Start comments with # – the rest of line is ignored.
- Can include a “documentation string” as the first line of any new function or class that you define.
- The development environment, debugger, and other tools use it: it’s good style to include one.

```
def my_function(x, y):
  """This is the docstring. This
  function does blah blah blah. """
  # The code would go here...
```

Assignment

- ***Binding a variable*** **in Python means setting a** ***name*** **to hold a** ***reference*** **to some** ***object*****.**
  - *Assignment creates references, not copies (like Java)*
- **A variable is created** ***the first time*** **it appears on the left side of an assignment expression:**

```
x = 3
```

- **An object is deleted (by the garbage collector) once it becomes unreachable.**
- ***Names in Python do not have an intrinsic type. Objects have types.***
  - Python determines the type of the reference automatically based on what data is assigned to it.

Multiple Assignment

- You can also assign to multiple names at the same time.

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

Naming Rules

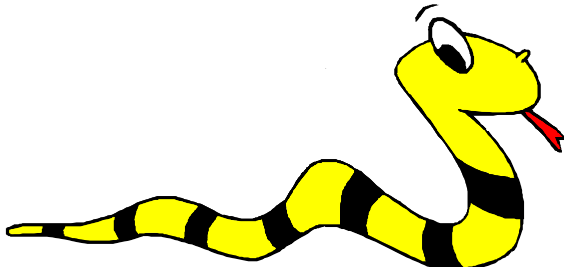
- **Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.**

  ```
  bob   Bob   _bob   _2_bob_   bob_2   BoB
  ```

- **There are some reserved words:**

  ```
  and, assert, break, class, continue, def, del,
  elif, else, except, exec, finally, for, from,
  global, if, import, in, is, lambda, not, or,
  pass, print, raise, return, try, while
  ```

Sequence types:
Tuples, Lists, and Strings



Sequence Types

1. Tulpes
   - A simple ***immutable*** ordered sequence of items
     - Immutable: a tuple cannot be modified once created....
   - Items can be of mixed types, including collection types

2. Strings
   - ***Immutable***
   - Conceptually very much like a tuple
   - [Py3] *UTF-8 Unicode* (type str)
   - [Py2] *8-bit chars* (type str)
     UTF-16 Unicode (type unicode)
3. Lists
   - ***Mutable*** ordered sequence of items of mixed types

Sequence Types 2

- The three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.

- Tuples are defined using parentheses (and commas).

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Lists are defined using square brackets (and commas).

```
>>> li = ["abc", 34, 4.34, 23]
```

- Strings are defined using quotes (", ', or """).

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line
string that uses triple quotes. """
```

Sequence Types 3

- We can access individual members of a tuple, list, or string using square bracket “array” notation.
- ***Note that all are 0 based…***

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]     # Second item in the tuple.
 'abc'

>>> li = ["abc", 34, 4.34, 23]
>>> li[1]      # Second item in the list.
 34

>>> st = "Hello World"
>>> st[1]   # Second character in string.
 'e'
```

Negative indices

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]
'abc'
```

Negative lookup: count from right, starting with –1.

```
>>> t[-3]
4.56
```

Slicing: Return Copy of a Subset - 1

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying ***before*** the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices when slicing.

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Optional argument allows selection of every nth item.

```
>>> t[1:-1:2]
('abc', (2,3))
```

Slicing: Return Copy of a Subset 2

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]
(4.56, (2,3), 'def')
```

Copying the Whole Sequence

To make a *copy* of an entire sequence, you can use `[:]`.

```
>>> t[:]
(23, 'abc', 4.56, (2,3), 'def')
```

Note the difference between these two lines for mutable sequences:

```
>>> list2 = list1     # 2 names refer to 1 ref
                      # Changing one affects both

>>> list2 = list1[:]  # Two independent copies, two refs
```

The ‘in’ Operator

- **Boolean test whether a value is inside a collection (often called a *container* in Python:**

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- **For strings, tests for substrings**

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- **Be careful: the *in* keyword is also used in the syntax of *for loops* and *list comprehensions*.**

The + Operator

- The + operator produces a ***new*** tuple, list, or string whose value is the concatenation of its arguments.

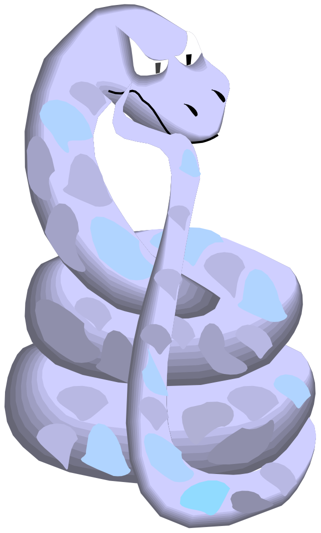
- Extends concatenation from strings to other types

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> "Hello" + " " + "World"
 'Hello World'
```

Mutability:
Tuples vs. Lists



Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
  ['abc', 45, 4.34, 23]
```

- **We can change lists *in place.***
- **Name *li* still points to the same memory reference when we're done.**

Tuples: Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

You can't change a tuple.
You can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

- ***The immutability of tuples means they're faster than lists.***

Operations on Lists Only - 1

```
>>> li = [1, 11, 3, 4, 5]

>>> li.append('a')  # Note the method syntax
>>> li
[1, 11, 3, 4, 5, 'a']

>>> li.insert(2, 'i')
>>>li
[1, 11, 'i', 3, 4, 5, 'a']
```

The *extend* method vs the + operator

- **+ creates a fresh list (with a new memory reference)**
- ***extend*** **is just like *add* in Java; it operates on list `li` in place.**

```
>>>li
[1, 11, 'i', 3, 4, 5, 'a']
>>> li.extend([9, 8, 7])
>>>li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

***Confusing*:**

- ***extend*** **takes a list as an argument**
- ***append*** **t akes a singleton as an argument, *unlike Java***

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operations on Lists Only - 3

```
>>> li = ['a', 'b', 'c', 'b']

>>> li.index('b')     # index of first occurrence*
1
```

*more complex forms exist

```
>>> li.count('b')     # number of occurrences
2

>>> li.remove('b')    # remove first occurrence
>>> li
  ['a', 'c', 'b']
```

Operations on Lists Only - 4

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()    # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()       # sort the list *in place*
>>> li
  [2, 5, 6, 8]

>>> li.sort(some_function)
    # sort in place using user-defined comparison
```

Tuples vs. Lists

- **Lists slower but more powerful than tuples.**
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.

- **To convert between tuples and lists use the list() and tuple() functions:**

```
li = list(tu)
tu = tuple(li)
```

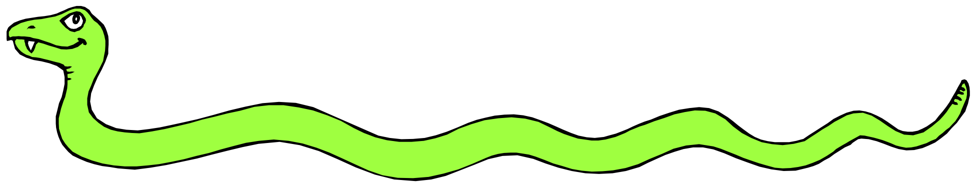
Sets, by examples

- Empty set: set()
- Indexing not supported
- Mixed types

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                            # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Dictionaries: a *mapping* collection type



Dictionaries: Like *maps* in Java

- **Dictionaries store a** *mapping* **between a set of keys and a set of values.**
  - ***Keys can be of any immutable hashable type***
    - ***cannot contain mutable components***
  - Values can be any type
  - Values and keys can be of different types in a single dictionary
- **You can**
  - define
  - modify
  - view
  - lookup
  - delete

  the key-value pairs in the dictionary.

Creating and accessing dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user']
'bozo'

>>> d['pswd']
1234

>>> d['bozo']

Traceback (innermost last):
  File ‘<interactive input>’ line 1, in ?
KeyError: bozo
```

- Keys must be unique.

```
>>> d1 = {1:7,1:5}
>>> d1
{1: 5}
```

Updating Dictionaries

- Assigning to an existing key changes the value.

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}
```

- Assigning to a non-existing key adds a new pair.

```
>>> d['id'] = 45
>>> d
{'user':'clown', 'id':45, 'pswd':1234}
```

- Dictionaries are unordered
  - New entry might appear anywhere in the output.
- (Dictionaries work by *hashing*)

Removing dictionary entries

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> del d['user']                # Remove one. Note that del is
                                # a function.
>>> d
{'p':1234, 'i':34}

>>> d.clear()                    # Remove all.
>>> d
{}
>>> a=[1,2]
>>> del a[1]                    # (del also works on lists)
>>> a
[1]
```

Useful Accessor Methods

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> list(d.keys())                  # List of current keys
['user', 'p', 'i']

>>> list(d.values())                # List of current values.
['bozo', 1234, 34]

>>> list(d.items())           # List of item tuples.
[('user','bozo'), ('p',1234), ('i',34)]
```

Using dictionaries

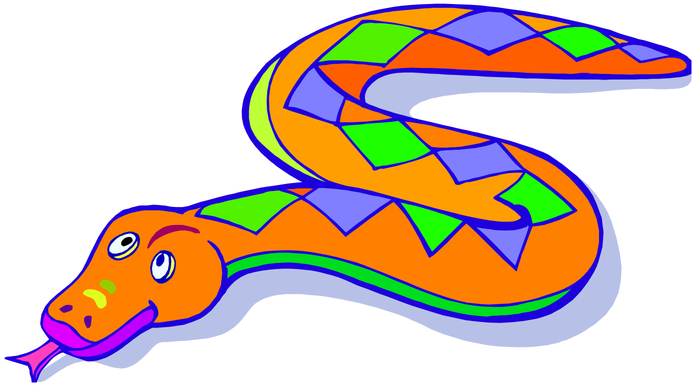
Write a program to compute the frequency of the words of a string read from the input. The output should print the words in increasing alphanumerical order.

```
freq = {}    # frequency of words in text [Python3]
line = input()
for word in line.split():
    freq[word] = freq.get(word,0)+1

words = list(freq.keys())
words.sort()

for w in words:
    print ("%s:%d" % (w,freq[w]))
```

Boolean Expressions



True and False

- ***True*** **and** ***False*** **are constants**
- ***Other values are treated as equivalent to either True or False when used in conditionals:***
  - *False*: zero, *None*, empty containers
  - *True*: non-zero numbers, non-empty objects
  - See PEP 8 for the most Pythonic ways to compare
- **Comparison operators: ==, !=, <, <=, etc.**
  - `X == Y`
    - X and Y have same value (like Java *equals* method)
  - `X is Y` :
    - X and Y refer to the *exact same object* (like Java ==)

Logical Operators

- **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`

Conditional Expressions

- `x = true_value if condition else false_value`
- lazy evaluation:
  - **First, `condition` is evaluated**
  - **If *True*, `true_value` is evaluated and returned**
  - **If *False*, `false_value` is evaluated and returned**

Control Flow

if Statements (as expected)

```
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'.")
```

Note:

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

while Loops (as expected)

```
>>> 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*

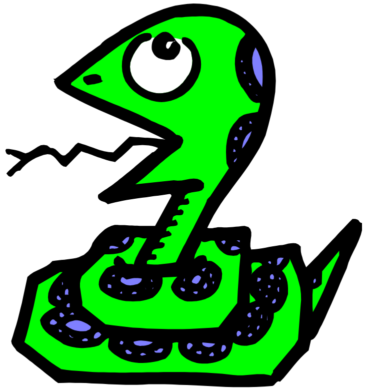
- You can use the keyword *break* inside a loop to leave the *while* loop entirely.
- You can use the keyword *continue* inside a loop to stop processing the current iteration of the loop and immediately go on to the next one.

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 throws an exception (**AssertionError**)

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

For Loops



For Loops 1

- For-each is Python's *only* form of for loop
- 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 complex than a single variable name.**
  - If the elements of <collection> are themselves collections, then <item> can match the structure of the elements. (We saw something similar with list comprehensions and with ordinary assignments.)

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

For loops and the *range()* function

- We often want to write a loop where the variables ranges over some sequence of numbers. The *range()* function returns a list of n numbers from 0 up to but not including the number we pass to it.
- range(5) returns [0,1,2,3,4]
- So we can say:

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

- Variant: range(start, stop[,step])

- [Py2]: *range()* returns a list, *xrange()* returns an *iterator* that provides the same functionality, more efficiently
- [Py3]: *range()* returns an iterator, *xrange()* illegal

Abuse of the *range()* function

- **Don't use *range()* to iterate over a sequence solely to have the index and elements available at the same time**

- **Avoid:**

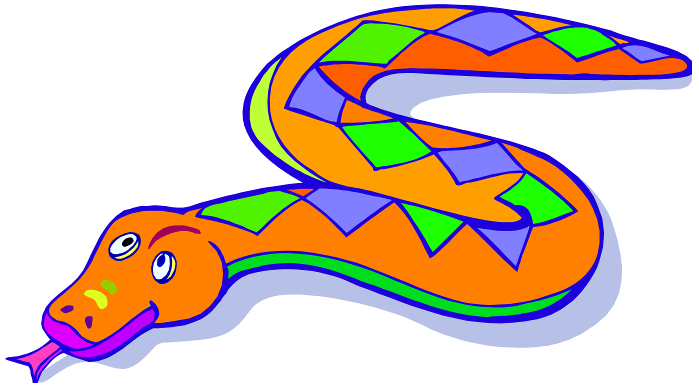
```
for i in range(len(mylist)):
    print(i, mylist[i])
```

- **Instead:**

```
for (i, item) in enumerate(mylist):
    print(i, item)
```

- **This is an example of an *anti-pattern* in Python**
  - For more, see: http://lignos.org/py_antipatterns/

Generating Lists using “List Comprehensions”



List Comprehensions 1

- **A powerful feature of the Python language.**
  - Generate a new list by applying a function to every member of an original list.
  - Python programmers use list comprehensions extensively. You'll see many of them in real code.

[expression for name in list]

[expression for name in list]

List Comprehensions 2

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

[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, and**
  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 3

- If the elements of list are other collections, then name can be replaced by a *collection* of names that match the “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 if filter]

Filtered List Comprehension 1

- Filter determines whether expression is performed on each member of the list.
- When processing each element of list, first check 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 2

```
>>> 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 produced.**

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].

For Loops / List Comprehensions

- **Python's list comprehensions provide a natural idiom that usually requires a for-loop in other programming languages.**
  - As a result, Python code uses many fewer for-loops

- *Caveat*! **The keywords *for* and *in* also appear in the syntax of list comprehensions, but this is a totally different construction.**