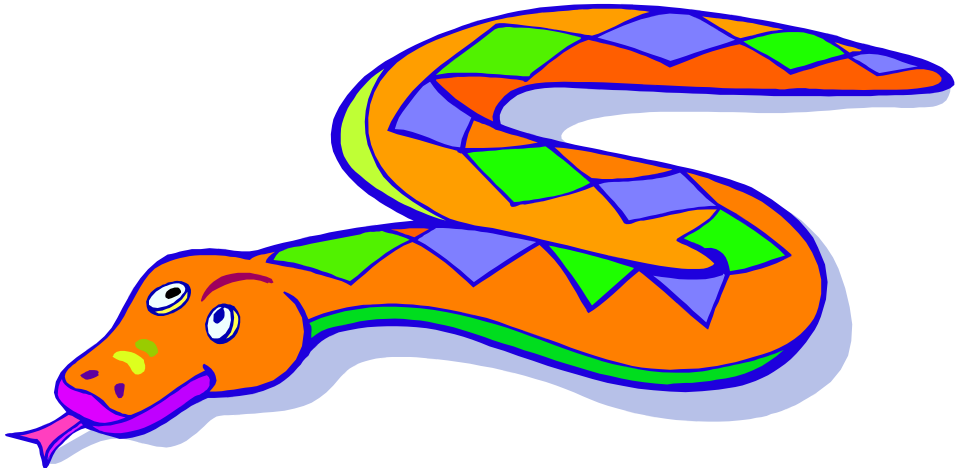
An Overview of Python



Brief History

- Created by Guido van Rossum in the late 1980s
- Named after ‘Monty Python’s Flying Circus’
- Python 2.0 was released on 16 October 2000
- Python 3.0 a major, backwards-incompatible release, was released on 3 December 2008
- Guido is the BDFL

Why do people use Python…?

The following primary factors cited by Python users seem to be these:

- Python is object-oriented

  Structure supports such concepts as polymorphism, operation overloading, and multiple inheritance.

- Indentation

  Indentation is one of the greatest future in

- Python. It's free (open source)

  Downloading and installing Python is free and easy

  Source code is easily accessible

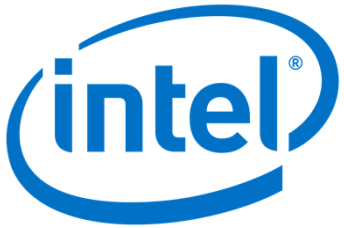
- It's powerful
  - Dynamic typing
    - Built-in types and tools
    - Library utilities
    - Third party utilities (e.g. Numeric, NumPy, SciPy)
  - Automatic memory management
- It's portable
  - Python runs virtually every major platform used today
  - As long as you have a compatible Python interpreter installed, Python programs will run in exactly the same manner, irrespective of platform.

It's mixable

- Python can be linked to components written in other languages easily
- Linking to fast, compiled code is useful to computationally intensive
  - problems
- - Python/C integration is quite common

- It's easy to use
  - No intermediate compile and link steps as in C/ C++
  - Python programs are compiled automatically to an intermediate form called *bytecode*, which the interpreter then reads
  - This gives Python the development speed of an interpreter without
  - the performance loss inherent in purely interpreted languages
- It's easy to learn
  - Structure and syntax are pretty intuitive and easy to grasp

Who uses python today…



Basic Datatypes

- Integers (default for numbers)

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

- Floats

  x = 3.456

- Strings

  Can use “” or ‘’ to specify. “abc” ‘abc’ (Same thing.)

  Unmatched ones 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 a new indentation is considered outside of the block.
  - Often a colon appears at the start of a new block.
    (We'll see this later for function and class definitions.)

```
>>> a = 1; b = 3
>>> if a > b:
...     result = 'bigger'
... elif a == b:
...     result = 'same'
... else:  # i.e. a < b
...     result = 'smaller'
...
>>> print(result)
smaller

>>> if a < b: print('ok')
ok
```

Identity Operators

There are two Identity operators explained below:

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

Operators Precedence

The following table lists all operators from highest precedence to lowest.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Calling a Function

- The syntax for a function call is:

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

- Parameters in Python are “Call by Assignment.”
  - Sometimes acts like “call by reference” and sometimes like “call by value” in C++. Depends on the data type.
  - We’ll discuss mutability of data types later: this will specify more precisely how function calls behave.

Names and References 1

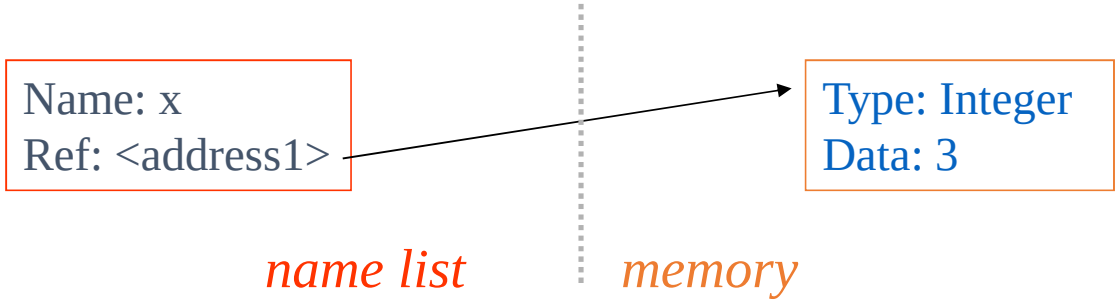
- Python has no pointers like C or C++. Instead, it has “names” and “references”. (Works a lot like Lisp or Java.)
- You create a name the first time it appears on the left side of an assignment expression:

```
x = 3
```

- Names store “references” which are like pointers to locations in memory that store a constant or some object.
  - Python determines the type of the reference automatically based on what data is assigned to it.
  - It also decides when to delete it via garbage collection after any names for the reference have passed out of scope.

Names and References 2

- There is a lot going on when we type:
  x = 3
- First, an integer 3 is created and stored in memory.
- A name x is created.
- An reference to the memory location storing the 3 is then assigned to the name x.



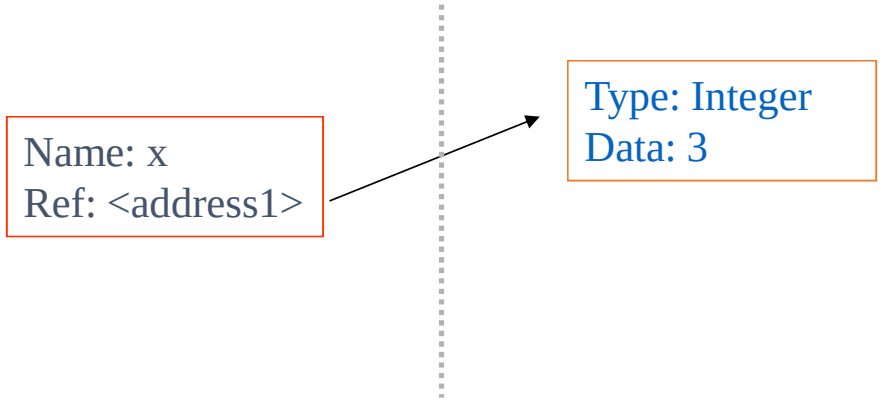
Names and References 3

- The data 3 we created is of type integer. In Python, the basic datatypes integer, float, and string are “immutable.”
- This doesn’t mean we can’t change the value of x… For example, we could increment x.

```
>>> x = 3
>>> x = x + 1
>>> print x
4
```

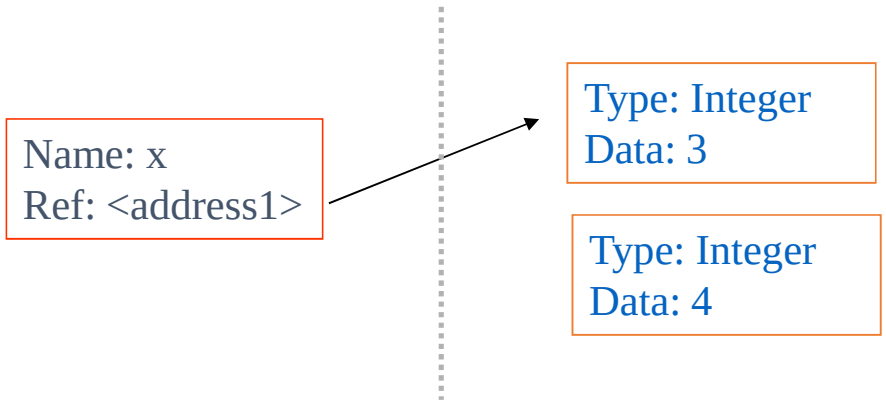
Names and References 4

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



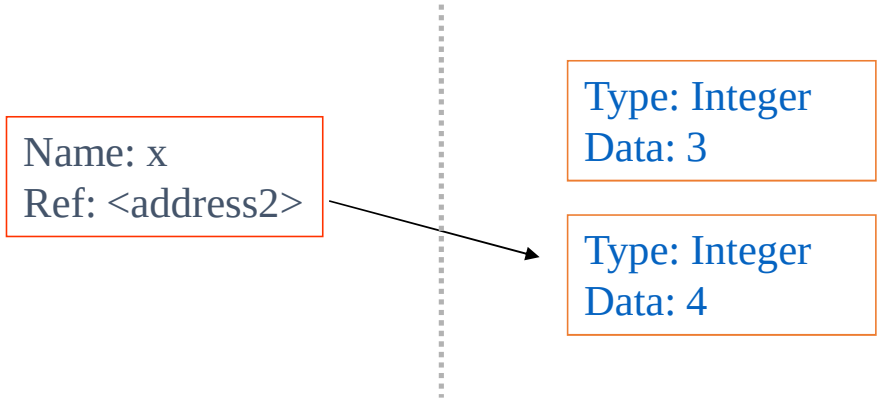
Names and References 4

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



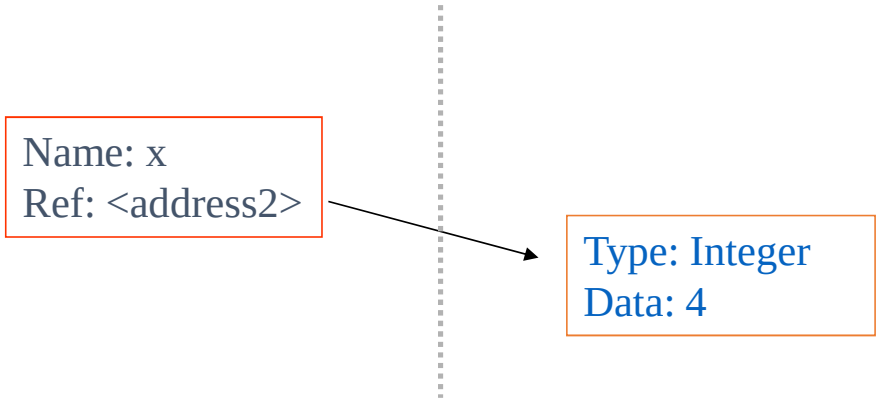
Names and References 4

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



Names and References 4

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



Container Types

- Last time, we saw the basic data types in Python: integers, floats, and strings.
- Containers are other built-in data types in Python.
  - Can hold objects of any type (including their own type).
  - There are three kinds of containers:

Tuples

- A simple immutable ordered sequence of items.

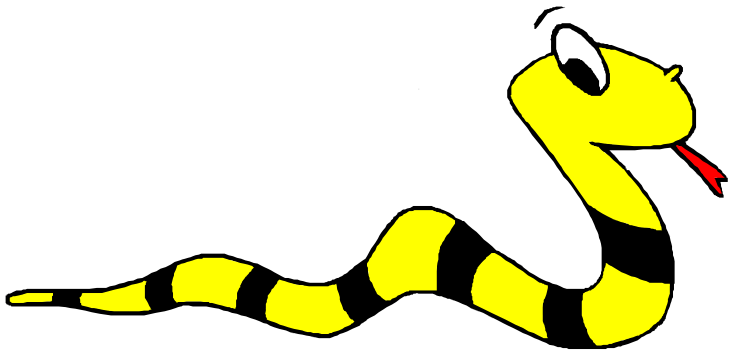
Lists

- Sequence with more powerful manipulations possible.

Dictionaries

- A look-up table of key-value pairs.

Tuples, Lists, and Strings: Similarities



Similar Syntax

- Tuples and lists are sequential containers that share much of the same syntax and functionality.
  - For conciseness, they will be introduced together.
  - The operations shown in this section can be applied to both tuples and lists, but most examples will just show the operation performed on one or the other.

- While strings aren't exactly a container data type, they also happen to share a lot of their syntax with lists and tuples; so, the operations you see in this section can apply to them as well.

Tuples, Lists, and Strings 1

- 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."""
```

Looking up an Item

```
>>> 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))
```

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 Container

You can make a copy of the whole tuple using **[:]**.

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

So, there's a difference between these two lines:

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

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

Membership Operators

There are two membership operators explained below:

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here **in** results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here **not in** results in a 1 if x is a member of sequence y.

The ‘in’ Operator

- Boolean test whether a value is inside a container:

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

- Be careful: the ‘in’ keyword is also used in the syntax of other unrelated Python constructions:
“for loops” and “list comprehensions.”

The + Operator

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

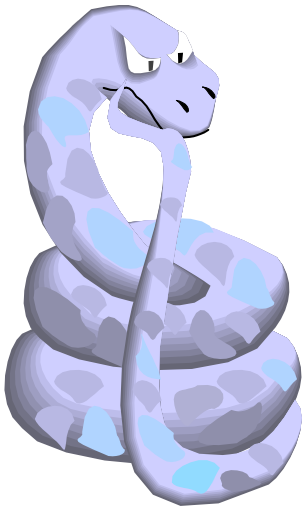
```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

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

>>> “Hello” + “ ” + “World”
 ‘Hello World’
```

The * Operator

- The * operator produces a new tuple, list, or string that “repeats” the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> “Hello” * 3
‘HelloHelloHello’
```

Mutability:
Tuples vs. Lists



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’re not allowed to change a tuple *in place* in memory; so, you can’t just change one element of it.

But it’s always OK to make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (1, 2, 3, 4, 5)
```

Lists: Mutable

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

We can change lists *in place.* So, it's ok to change just one element of a list. Name li still points to the same memory reference when we're done.

List Methods

Method	Description
Nonmutating methods	
`L.count(x)`	Returns the number of items of `L` that are equal to `x`.
`L.index(x)`	Returns the index of the first occurrence of an item in `L` that is equal to `x`, or raises an exception if `L` has no such item.
Mutating methods	
`L.append(x)`	Appends item `x` to the end of `L`; e.g., `L[len(L):]=[x]`.
`L.extend(s)`	Appends all the items of iterable `s` to the end of `L`; e.g., `L[len(L):]=s`.
`L.insert(i, x)`	Inserts item `x` in `L` before the item at index `i`, moving following items of `L` (if any) "rightward" to make space (increases `len(L)` by one, does not replace any item, does not raise exceptions: acts just like `L[i:i]=[x]`).
`L.remove(x)`	Removes from `L` the first occurrence of an item in `L` that is equal to `x`, or raises an exception if `L` has no such item.
`L.pop([i])`	Returns the value of the item at index `i` and removes it from `L`; if `i` is omitted, removes and returns the last item; raises an exception if `L` is empty or `i` is an invalid index in `L`.
`L.reverse( )`	Reverses, in place, the items of `L`.
`L.sort([f])` (2.3)	Sorts, in place, the items of `L`, comparing items pairwise via function `f`; if `f` is omitted, comparison is via the built-in function `cmp`. For more details, see "Sorting a list" on page 57.
`L.sort(cmp=cmp, key=None, reverse=False)`(2.4)	Sorts, in-place, the items of `L`, comparing items pairwise via the function passed as *cmp* (by default, the built-in function `cmp`). When argument *key* is not `None`, what gets compared for each item `x` is `key(x)`, not `x` itself. For more details, see "Sorting a list" on page 57.

Slicing: with mutable lists

- >>> L = [‘spam’, ’Spam’, ’SPAM’]

  >>> L[1] = ‘eggs’

  >>> L

  [‘spam’, ‘eggs’, ‘SPAM’]

- >>> L[0:2] = [‘eat’,’more’]

  >>> L

  [‘eat’, ‘more’, ‘SPAM’]

Operations on Lists Only 1

- Since lists are mutable (they can be changed in place in memory), there are many more operations we can perform on lists than on tuples.

- The mutability of lists also makes managing them in memory more complicated… So, they aren't as fast as tuples. It's a tradeoff.

Operations on Lists Only 2

```
>>> li = [1, 2, 3, 4, 5]

>>> li.append('a')
>>> li
[1, 2, 3, 4, 5, 'a']

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

NOTE: li = li.insert(2,'I') loses the list!

Operations on Lists Only 3

The ‘extend’ operation is similar to concatenation with the + operator. But while the + creates a fresh list (with a new memory reference) containing copies of the members from the two inputs, the extend operates on list **`li`** in place.

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

Extend takes a list as an argument. Append takes a singleton.

```
>>> li.append([9, 8, 7])
>>> li
[1, 2, ‘i’, 3, 4, 5, ‘a’, 9, 8, 7, [9, 8, 7]]
```

Operations on Lists Only 4

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

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

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

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

Operations on Lists Only 5

```
>>> 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
```

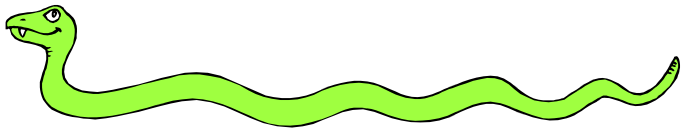
Tuples vs. Lists

- 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.

- We can always convert between tuples and lists using the list() and tuple() functions.

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

Dictionaries



Basic Syntax for Dictionaries 1

- Dictionaries store a mapping between a set of keys and a set of values.
  - Keys can be any immutable type.
  - Values can be any type, and you can have different types of values in the same dictionary.
- You can define, modify, view, lookup, and delete the key-value pairs in the dictionary.

Basic Syntax for Dictionaries 2

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

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

>>> d['pswd']
1234

>>> d['bozo']

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

Basic Syntax for Dictionaries 3

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

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

Note: Keys are unique.
Assigning to an existing key just replaces its value.

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

Note: Dictionaries are unordered.
New entry might appear anywhere in the output.

Basic Syntax for Dictionaries 4

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

>>> del d['user']              # Remove one.
>>> d
{'p':1234, 'i':34}

>>> d.clear()                  # Remove all.
>>> d
{}
```

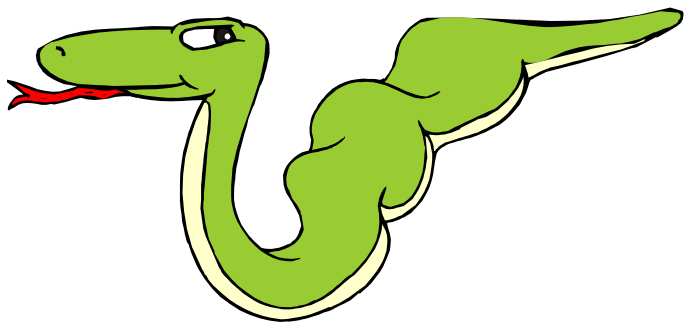
Basic Syntax for Dictionaries 5

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

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

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

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

Dictionary Methods

Method	Description
Nonmutating methods	
`D.copy()`	Returns a shallow copy of the dictionary (a copy whose items are the same objects as *D*'s, not copies thereof)
`D.has_key(k)`	Returns `True` if *k* is a key in *D*; otherwise, returns `False`, just like *k* `in` *D*
`D.items()`	Returns a new list with all items (key/value pairs) in *D*
`D.keys()`	Returns a new list with all keys in *D*
`D.values()`	Returns a new list with all values in *D*
`D.iteritems()`	Returns an iterator on all items (key/value pairs) in *D*
`D.iterkeys()`	Returns an iterator on all keys in *D*
`D.itervalues()`	Returns an iterator on all values in *D*
`D.get(k[,x])`	Returns *D*[*k*] if *k* is a key in *D*; otherwise, returns *x* (or `None`, if *x* is not given)
Mutating methods	
`D.clear()`	Removes all items from *D*, leaving *D* empty
`D.update(D1)`	For each *k* in *D1*, sets *D*[*k*] equal to *D1*[*k*]
`D.setdefault(k[,x])`	Returns *D*[*k*] if *k* is a key in *D*; otherwise, sets *D*[*k*] equal to *x* and returns *x*
`D.pop(k[,x])`	Removes and returns *D*[*k*] if *k* is a key in *D*; otherwise, returns *x* (or raises an exception if *x* is not given)
`D.popitem()`	Removes and returns an arbitrary item (key/value pair)

Assignment and Containers



Multiple Assignment with Container Classes

- We've seen multiple assignment before:

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

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

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