Objects

• Everything in Python is an "object"

• A “thing” with a type and a value

• All objects have an object id
  • Available via the id() function
Object IDs

- How Python internally keeps track of all entities

```python
>>> id(1)
4298185464
>>> x = 1
>>> id(x)
4298185464
>>> y = x
>>> id(y)
4298185464
>>> y = "one"
>>> id(y)
4299778088
>>> id(len)
4297684536
>>> id(list)
4296508768
>>> id(list.append)
4297566832
>>> list.append
<method 'append' of 'list' objects>
```
Object Identity

* The **is** operator tests whether two variables or expressions refer to the same object

```python
def compare(a, b):
    if a is b:
        # a and b are the same object
        <statements>
    if a == b:
        # a and b have the same value
        <statements>
    if type(a) is type(b):
        # a and b have the same type
        <statements>

if type(s) is list:
    s.append(item)
if type(d) is dict:
    d.update(t)
```
“is-a” Testing

- The `isinstance` operator

```python
if isinstance(s, list):
    s.append(item)
if isinstance(d, dict):
    d.update(t)

>>> class A(object): pass
...
>>> class B(A): pass
...
>>> b = B()
>>> isinstance(b, A)
True
>>> isinstance(B, A)
False
>>> A in B.__bases__
True
```
Reference Counting

- Every variable binding to an object increments its reference count
- When the variable goes out of scope, the count is decremented
  - Or when it is deleted with `del`
- When the count reaches zero, the object is eligible for garbage collection

```python
import sys
print sys.getrefcount(37)  # 17
a = 37
print sys.getrefcount(37)  # 18
b = a
print sys.getrefcount(37)  # 19
c = []
c.append(b)
print sys.getrefcount(37)  # 20
del a
print sys.getrefcount(37)  # 19
b = 42
print sys.getrefcount(37)  # 18
c[0] = 2.0
print sys.getrefcount(37)  # 17
a = 37
print sys.getrefcount(37)  # 18
```
References and Copies

- Statements such as \( a = b \) make no copies of data.
- The variable \( a \) simply refers to the same object \( b \) refers to.
  - The \texttt{id} (address) of \( b \)'s object is bound to \( a \).
- This is part of the “everything is an object” feature of Python.
- Copies of data are made only when \textit{requested}.
  - And usually are only an issue with \textit{containers}.
Copies and Containers

- You have 3 choices in “copying” containers (sequences & dictionaries)
  
- 1) \( y = x \)  
  (No copy made)

- 2) \( y = \text{list}(x) \) (or \( \text{tuple}(x) \), \( \text{set}(x) \), ...; does shallow copy)

- 3) import copy
  \( y = \text{copy}.\text{deepcopy}(x) \)
The Chuckster Making Copies

```python
>>> a = [1, 2, [3, 4]]
>>> b = list(a) # Create a shallow copy of a. Also: “b = a[:]”
>>> b is a
False
>>> a == b  # “value” equality
True
>>> b.append(100) # Append element to b.
>>> b
[1, 2, [3, 4], 100]
>>> a # Notice that a is unchanged
[1, 2, [3, 4]]
>>> b[2][0] = -100 # Modify an element inside b
>>> b
[1, 2, [-100, 4], 100]
>>> a # Notice the change inside a
[1, 2, [-100, 4]]
```
Deep Copies

>>> import copy
>>> a = [1, 2, [3, 4] ]
>>> b = copy.deepcopy(a)
>>> b[2][0] = -100
>>> b
[1, 2, [-100, 4] ]
>>> a # Notice that a is unchanged
[1, 2, [3, 4] ]
>>> part = [1,2]
>>> whole = [part, part]
>>> shallow = whole[:]
>>> deep = copy.deepcopy(whole)
>>> del part[0]
>>> part
[2]
>>> whole
[[2], [2]]
>>> shallow
[[2], [2]]
>>> deep
[[1, 2], [1, 2]]
>>> del deep[0][0]
>>> deep
[[2], [2]]
part

[ * , * ]

[ * , * ] whole

shallow

[ * , * ]

1

2

[ * , * ]

deep

[ * , * ]

[ * , * ]
Everything Really Is an Object

```python
items = {}
items["func"] = abs
import math
items["mod"] = math # Add a module
items["error"] = ValueError # Add an exception type
nums = [1, 2, 3, 4]
items["append"] = nums.append # Add a method bound to an object
>>> items["func"](-45) # Executes abs(-45)
45
>>> items["mod"]).sqrt(4) # Executes math.sqrt(4)
2.0
>>> try:
. . .     x = int("a lot")
. . . except items["error"]: # Same as except ValueError
. . .     print("Couldn't convert")
. . .
Couldn't convert
>>> items["append"](100) # Executes nums.append(100)
>>> nums
[1, 2, 3, 4, 100]
```
zip and split

* zip() combines multiple lists into lists of tuples
* split() returns a list of separated substrings

```python
>>> list1
[1, 2, 3]
>>> list2
['one', 'two', 'three', 'four']
>>> list3
['bob', 'carol', 'test', 'alice']
>>> zip(list1, list2, list3)
[(1, 'one', 'bob'), (2, 'two', 'carol'), (3, 'three', 'test')]
>>> s = 'now is the time'
>>> s.split()
['now', 'is', 'the', 'time']
>>> s = 'one,two,three,four'
>>> s.split(',
[one', 'two', 'three', 'four']
```
A First Look at List Comprehensions

* Easy way to define a list
* Put an expression on a loop variable in brackets

```python
>>> [x*x for x in range(1,5)]
[1, 4, 9, 16]
>>> [str(x)+'!' for x in list2]
['one!', 'two!', 'three!', 'four!']
>>> [str(x)+'!' for x in list2 if len(str(x)) > 3]
['three!', 'four!']
```
Powerful, Compact Code!

```python
type = "GOOG,100,490.10"
field_types = [str, int, float]
raw_fields = type.split(',
print raw_fields
print zip(field_types,raw_fields)
fields = [ty(val) for ty,val in zip(field_types,raw_fields)]
print fields

''' Output:
['GOOG', '100', '490.10']
[(<type 'str'>, 'GOOG'), (<type 'int'>, '100'), (<type 'float'>, '490.10')]
['GOOG', 100, 490.10000000000002]'''
```
# Built-in Types

<table>
<thead>
<tr>
<th>Type Category</th>
<th>Type Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>NoneType</td>
<td>The null object</td>
</tr>
<tr>
<td>Numbers</td>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>Arbitrary-precision Integer (2.x only)</td>
</tr>
<tr>
<td></td>
<td>float</td>
<td>64-bit IEEE Numbers (&quot;float&quot;)</td>
</tr>
<tr>
<td></td>
<td>complex</td>
<td>Scary Numbers :-)</td>
</tr>
<tr>
<td></td>
<td>bool</td>
<td>True or False</td>
</tr>
<tr>
<td>Sequences</td>
<td>str</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unicode</td>
<td>(2.x only)</td>
</tr>
<tr>
<td></td>
<td>list</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tuple</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xrange</td>
<td>Range iterator (range in 3.x)</td>
</tr>
<tr>
<td>Mapping</td>
<td>dict</td>
<td></td>
</tr>
<tr>
<td>Sets</td>
<td>set</td>
<td>Mutable Set</td>
</tr>
<tr>
<td></td>
<td>frozenset</td>
<td></td>
</tr>
</tbody>
</table>
Slices

* Extracting *subsets* of a sequence by *index*

* `a[1:4]`  # returns a list containing `a[1]` through `a[3]`

* `a[3:]`  # returns `[a[3], a[4], ...]`

* `a[:5]`  # returns `[a[0], a[1], a[2], a[3], a[4]]`

* `a[-1]`  # returns `a[len(a)-1]`

* `a[1::2]`  # returns `[a[1], a[3], a[5], ...]`

* `a[-n]`  # returns `a[len(a)-n]`
Envisioning Slices

- "Help"
  +---+---+---+---+
  | H | e | l | p |
  +---+---+---+---+
  0   1   2   3   4 (outside)
  -4  -3  -2  -1

- The first (0-th) character is \texttt{s[-len(s)]}

- All sequences can be indexed with slices
Strides

- A third slice parameter defines a stride:
  - The amount to skip between elements of interest
  - `[start:end+1:stride]`

- `x[m:n:s]` yields `x[m]`, `x[m+s]`, …, `<x[n]`

- If stride is negative, you need `m > n` to get a non-empty sequence: `x[m]`, `x[m-s]`, …

- `a[5:1:-2]`  # returns `[a[5],a[3]]`

- `a[::-1]`  # returns the list `a` reversed
Slice “Quiz”

• Given a = “hello”:

```python
a[::2] == ?
a[:::-2] == ?
a[::2:-1] == ?
a[5:0:-1] == ?
a[0:10:-1] == ?
a[-2:10] == ?
a[10::2] == ?
```
Slice Assignment

```
x = [1, 2, 3, 4]
x[1:3] = [22, 33, 44]
x
[1, 22, 33, 44, 4]
x[1:4] = [2, 3]
x
[1, 2, 3, 4]
x[1:1] = (100, 200) # Note tuple
x
[1, 100, 200, 2, 3, 4]
x[1:2] = []
x
[1, 200, 2, 3, 4]
del x[1]
x
[1, 2, 3, 4]
```
Slice Assignment with Strides

```python
>>> x = [1, 2, 3, 4]
>>> x[::2] = ['x', 'y']  # Same as x[0:4:2] = ...
>>> x
['x', 2, 'y', 4]
>>> x[::2] = ['x', 'y', 'z']
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: attempt to assign sequence of size 3 to extended slice of size 2

Sizes must match!
```
String Methods

- capitalize
- center
- count
- endswith
- find
- index
- isalpha
- isdigit
- istitle
- join
- lfind
- ljust
- lstrip
- lower
- replace
- rfind
- rjust
- rstrip
- split
- startswith
- strip
- swapcase
- title
- translate
- upper
- zfill
Dictionary Methods

- if k in d: ... (key search)
- keys() (list of keys)
- values() (list of values)
- items() (list of pairs)
- copy() (shallow copy)
- dict.fromkeys(<seq>[,val])
- d1.update(d2) (merge 2 dicts)
- get(key, def = None)
- setdefault() (= get() + create)
- pop(key [, def])
- Can also use del to remove elements by key
- popitem()
- clear()
Dictionary Iterators

* You can *iterate* over the items (pairs), keys, or values of a dictionary

* for p in d.iteritems(): ...

* for k in d.iterkeys(): ...

* for v in d.itervalues(): ...

* Remove the “iter” prefix in 3.x
Set Example

```python
>>> a = set(range(5))    # same as a = {0,1,2,3,4}
>>> a
set([0, 1, 2, 3, 4])
>>> b = set(range(3,10))
>>> b
set([3, 4, 5, 6, 7, 8, 9])
>>> a.union(b)                 # or 
a | b
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a.intersection(b)          # or 
a & b
set([3, 4])
>>> a.difference(b)            # or 
a - b
set([0, 1, 2])
>>> a.symmetric_difference(b)  # or 
a ^ b
set([0, 1, 2, 5, 6, 7, 8, 9])
>>> a.issuperset(set([1,2,3])) # or 
a >= b
True
>>> c = a.copy()
>>> c.discard(3)
>>> c.issubset(a)              # or 
c <= a or c < a
True
>>> c.update(a)
>>> c
set([0, 1, 2, 3, 4])
```

See Tables 3.7 and 3.8
Callable Types

- Functions
- Methods
- Objects (by defining a `__call__( )` method)
Standard Function Attributes

- `__doc__`: Documentation string
- `__name__`: Function name
- `__dict__`: Dictionary containing function attributes
- `__code__`: Byte-compiled code
- `__defaults__`: Tuple containing the default arguments
- `__globals__`: Dictionary defining the global namespace
- `__closure__`: Tuple containing data related to nested scopes

See counter*.py
Types of Methods

- **Instance Methods**
  - Apply to an *instance* of a class

- **Class Methods***
  - Apply to the one and only *class object* (see Chapter 7)

- **Static Methods**
  - Normal functions *(no self)* defined inside a class

- **All can be bound** *(to an instance or class)* or *unbound*

  * Class methods are quite unique; no equivalent in Java, C#, C++, ...
Types of Methods

class Foo(object):
    def instance_method(self, args...):
        <statements>

@classmethod
def class_method(cls, args...):
    <statements>

@staticmethod
def static_method(args...):
    <statements>
Counting Instances of a Class

• An illustration of static data and methods:
  • Initialize a static variable to 0
  • Increment it in the constructor

• See counted*.py
Overloading __call__

- An instance method

- Similar to overloading `operator()` in C++ and D

- See `call.py`

- See Table 3.19 in the book for common overloadable operations
Bound Methods

- An instance method can be packaged with an object resulting in a new callable entity
- Likewise for static and class methods with class objects
- Similar to delegates in C# and D
- Useful for callbacks from frameworks (e.g., event handlers)
- See bound*.py, unbound*.py
Bound Methods in a GUI

from Tkinter import *

class App:
    def __init__(self, master):
        frame = Frame(master)
        frame.pack()

        self.button = Button(frame, text="QUIT", fg="red", command=frame.quit)
        self.button.pack(side=LEFT)

        self.hi_there = Button(frame, text="Hello", command=self.say_hi)
        self.hi_there.pack(side=LEFT)

    def say_hi(self):
        print "hi there, everyone!"

root = Tk()
app = App(root)
root.mainloop()