

Python Tutorial

Day 3

Agenda

- classes and object orientation
- some modules
- SIDB: The **Silly Inventory Database**
- modules: serialization, regular expressions
- list comprehensions

with Class

- a `class` block defines a type: what's in it, what it can do
- classes may be derived from other classes
- each instance of that class knows what it can do
- each instance has a private namespace
- special methods for under-the-hood trickery, indicated `__special__`

Self-aware

- each instance has a copy of local variables
- to access the private namespace, as always, use dot notation

```
import math
class Point:
    def distance(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)
# ---
>>> v = Point()
>>> v.x, v.y = 3, 5
>>> v.distance()
5.8309518948453007
```

Initializing

- most common special method is `__init__`
- called when creating a new instance, `classname()`
- most common task: setting instance variables

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)
```

Refine Distance

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, p2):
        if isinstance(p2, Point):
            return math.sqrt((self.x - p2.x) ** 2 + (self.y - p2.y) ** 2)
        else:
            raise TypeError, "'%s' not type Point" % p2
#-----
>>> v = Point(3, 2)
>>> v.distance(Point(3, 7))
5.0
>>> v.distance(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 9, in distance
TypeError: '3' not type Point
>>>
```

Subclassing

- a subclass knows everything the class knows
- but you can override certain behaviors
- or you may add new ones
- a common empty subclass: new exceptions with appropriate names

```
class PointTypeError(TypeError):  
    pass
```

SIDB: The Silly Inventory Database

- this is an absurd application, but convenient for demonstration
- we will keep track of desktop computers
- will assign ownership using Unix login
- store in a file
- do basic searches

InventoryItem

```
class InventoryItem:
    def __init__(self, hostname, opsys, user, location):
        self.hostname = hostname
        self.os = opsys
        self.user = user
        self.location = location
# -----
>>> f = InventoryItem('cydonia', 'osx', 'annis', 'J4/503')
>>> f
<__main__.InventoryItem instance at 0x6e9b8>
```

Nicer Output

- special method: `__str__`

```
def __str__(self):
    return "%s (%s), used by %s in %s." % (self.hostname, self.os,
                                          self.user, self.location)

# ----
>>> f = InventoryItem('cydonia', 'osx', 'annis', 'J4/503')
>>> f
<__main__.InventoryItem instance at 0x6e9b8>
>>> print f
cydonia (osx), used by annis in J4/503.
```

- define `__repr__` to change interactive representation

Even Nicer Output, Part the First

- using Unix login names is sort of impersonal
- we can get fuller information from the Unix passwd database

```
>>> import pwd
>>> pwd.getpwnam('annis')
('annis', 'o2i6tha002y8A', 582, 100, 'William Annis',
  '/u/annis', '/bin/bash')
>>> pwd.getpwnam('anniss')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 'getpwnam(): name not found: anniss'
>>>
```

Even Nicer Output, Part the Second

```
def __str__(self):
    try:
        fullname = pwd.getpwnam(self.user)[4]
    except KeyError:
        fullname = "unknown user '%s'" % self.user

    return "%s (%s), used by %s in %s." % (self.hostname, self.os,
                                          fullname, self.location)

# -----
>>> f = InventoryItem('cydonia', 'osx', 'annis', 'J4/503')
>>> print f
cydonia (osx), used by William Annis in J4/503.
>>> g = InventoryItem('cydonia2', 'osx', 'anniss', 'J4/505')
>>> print g
cydonia2 (osx), used by unknown user 'anniss' in J4/505.
```

InventoryDB

- create instance with filename argument
- if file exists, read as existing DB
- if no file yet, it's a new DB
- selection match by regular expression, or by string equality

InventoryDB: File

- python has a built in way to *serialize* objects
- saves you having to reinvent a file format for every program
- python serialized data called a “pickle”
- normally use `cPickle` module (faster)
- `dumps/loads` work with strings

- dump/load work with files
- when you load a pickle for a class, your program should already know about class definition
- pickle format not nice for humans to read
- there is also a binary pickle format (saves space)

os module

- a bunch of OS functions are in `os` module: process manipulation, pipes, directory ops, `fork`, `stat`, etc.
- submodule, loaded automatically, `os.path` has many useful functions
- `os.path.basename(f)` returns only file name
- `os.path.dirname(f)` returns only directory name
- `os.path.exists(f)` check if file exists

- `os.path.isdir(f)`, `ispath(f)`, `islink(f)`: check file type
- `os.path.getctime(f)` checks file change-time
- many others; do `dir(os.path)` for full list

InventoryDB: init

```
class InventoryDB:
    def __init__(self, dbfile):
        self.dbfile = dbfile
        # DB itself is list of InventoryItem instances.
        self.db = []
        # Don't save if not necessary.
        self.need_to_save = False

        if os.path.exists(dbfile):
            self.loaddb()
```

```
def loaddb(self):
    f = open(self.dbfile, "r")
    self.db = cPickle.load(f)
    f.close()

def savedb(self):
    if self.need_to_save:
        f = open(self.dbfile, "w")
        cPickle.dump(self.db, f)
        f.close()
        self.need_to_save = False
```

Adding Items

```
# Local exceptions:
class SIDBTypeError(TypeError):
    pass

#...

def add(self, item):
    if not isinstance(item, InventoryItem):
        raise SIDBTypeError, "'%s' not InventoryItem" % item
    else:
        self.db.append(item)
        self.need_to_save = True
```

```
>>> db = InventoryDB("stuff")
>>> db.add('computah')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "sidb", line 63, in add
    raise SIDBTypeError, "'%s' not InventoryItem" % item
__main__.SIDBTypeError: 'computah' not InventoryItem
>>> db.add(InventoryItem('cydonia', 'osx', 'annis', 'J4/503'))
>>> db.add(InventoryItem('wazor', 'solaris', 'annis', 'J4/503'))
>>> db.add(InventoryItem('hydra', 'solaris', 'system', 'server room'))
>>> db.add(InventoryItem('centauri', 'solaris', 'system', 'server room'))
>>> db.savedb()
```

Diversion: List Comprehensions

- very often we iterate over lists to make new lists
- we might **filter** or we might **transform**
- python has a syntax for simple cases of both operations
- when you see what looks like a `for` loop inside square brackets, that's a list comprehension

Comprehensions: Filter

- [EXPR for VAR in LISTVAR if ...]

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> [i for i in x if i > 4]
[5, 6, 7, 8]
>>> [i*2 for i in x if i > 4]
[10, 12, 14, 16]
>>> [math.sqrt(i) for i in x if i > 4]
[2.2360679774997898, 2.4494897427831779,
 2.6457513110645907, 2.8284271247461903]
>>>
```

- for myself, I say “collecting $i*2...$ ” when I see a comprehension

Comprehensions: Transform

- easy, just remove if clause

- [EXPR for VAR in LISTVAR]

```
>>> l = "This IS NeaTO fun".split()
>>> l
['This', 'IS', 'NeaTO', 'fun']
>>> [word.lower() for word in l]
['this', 'is', 'neato', 'fun']
>>>
```

- if the transformation is complex, write a function for EXPR

Object Attributes

- the names of instance variables and methods are called **attributes**
- so, our InventoryItem class has `__init__`, `user`, `hostname`, etc. attributes
- So how do you check if an instance has certain attributes?
- if you access it, and it doesn't exist, `AttributeError` will be raised

Attribute Testing

- `hasattr` returns boolean

```
>>> f = InventoryItem('cydonia', 'osx', 'annis', 'J4/503')
>>> hasattr(f, 'os')
True
>>> hasattr(f, 'spam')
False
>>> hasattr(f, '__init__')
True
>>>
```

Attribute Retrieval

- `getattr` tries to retrieve data, with default fall-back available

```
>>> f = InventoryItem('cydonia', 'osx', 'annis', 'J4/503')
>>> getattr(f, 'os')
'osx'
>>> getattr(f, 'spam')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: InventoryItem instance has no attribute 'spam'
>>> getattr(f, 'spam', "eggs")
'eggs'
>>>
```

Selecting Exact Matches

```
def select_exact(self, attr, val):
    return [m for m in self.db if getattr(m, attr) == val]

# ----
>>> db = InventoryDB("stuff")
>>> for a in db.select_exact('os', 'solaris'):
...     print a
...
wazor (solaris), used by William Annis in J4/503.
hydra (solaris), used by unknown user 'system' in server room.
centauri (solaris), used by unknown user 'system' in server room.
>>>
```

Regular Expressions

- unlike perl, regular expressions (regexps) have no special syntax in python (i.e., no `a =~ s/hte/the/g;`)
- the module `re` is a regexp library which uses notation like perl
- almost always you will want to use the `re.search` function, which returns a `SRE_Match` instance, or `None` on failure
- `re.match` only returns true when the **entire string** matches correctly (verbose to do right)

Regular Expressions

- fuller discussion next time
- character classes: `[aeiou]` matches any **one** of the characters between square brackets
- `.` matches anything, `.*` matches zero or more instances of anything, `.+` one or more
- `\s` is whitespace, `\w` a “word” character, `\d` a digit

```
>>> spam = "On 3/9/2006 William gibbers about Python."
>>> f = re.search("\d/\d/\d*", spam)
>>> f
<_sre.SRE_Match object at 0x50d78>
>>> f.group()
'3/9/2006'
>>> f = re.search("Gibbers", spam)
>>> f
>>> f = re.search("Gibbers", spam, re.IGNORECASE)
>>> f
<_sre.SRE_Match object at 0x50d78>
>>> f.group()
'gibbers'
>>>
```

InventoryDB: Search by Pattern

```
def select_match(self, attr, pattern):
    answ = []
    for item in self.db:
        match = re.search(pattern, getattr(item, attr), re.IGNORECASE)
        if match is not None:
            answ.append(item)
    return answ

# ----
>>> for a in db.select_match('hostname', "c[yi]d.+"):
...     print a
...
cydonia (osx), used by William Annis in J4/503.
>>>
```

In Two Weeks

- reminder: I'll be out next week so no class
- more regular expressions
- refine the InventoryDB using more special methods to clean up code
- writing your own libraries
- more modules (not sure which yet)