Privacy in Python

- There is no true privacy.
- Almost private: put __ in front of any name

```
__var becomes _class__var

class MyClass(object):
    def __fun(self):
        print(100)

my_inst = MyClass()
my_inst.__fun()          # ERROR
my_inst._MyClass__fun()  # prints 100
```
Qualified namespaces are dicts

- Every object and module has a namespace
- The namespace’s dictionary is bound to a special variable `__dict__`
  - Lists local values (variables, functions, etc.) in the object

Example class namespace

```python
class MyClass(object):
    x = 100
    def my_method(param1=500):
        print(param1)

>>> for k, v in MyClass.__dict__.items():
    print(k, v)
__module__ __main__
__dict__ <attribute '__dict__' of 'MyClass' objects>
x 100
my_method <function my_method at 0x1029b4e20>
__weakref__ <attribute '__weakref__' of 'MyClass' objects>
__doc__ None
```
Example instance namespace

```python
class MyClass(object):
    x = 100
    def my_method(param1=500):
        print(param1)

>>> my_inst = MyClass()
>>> for k,v in my_inst.__dict__.items():
    print(k,v)
    # nothing in L-scope namespace!
>>> my_inst.x = 5  # L-scope assignment
>>> for k,v in my_inst.__dict__.items():
    print(k,v)
    x 5
```

Michigan State University
CSE 231, Fall 2013

---

collection methods
If you want your object to act as a container (such as a list, set or dictionary), you can overload operators for your container.

What operators would those be?

---

**Python container mappings**

- `item in x` maps to `x.__contains__(item)`
- `len(x)` maps to `x.__len__(self)`
- `x[key]` maps to `x.__getitem__(key)`
- `x[key] = value` maps to `x.__setitem__(key, value)`
more functions

__delitem__(self, key)
    remove key from instance (if mutable)

__iter__(self)
    return an iterator on the instance

Expansion class
Represent \( \pi \) as an infinite sequence of values.
Let us code a sequence to represent \( \pi \)
and code it as a Python class.

\[
\frac{\pi}{4} = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{2k - 1}
\]

\[
\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \ldots
\]
Use our rational class

We can represent $\pi$ as a sequence of rational numbers that, when added together, approximate $\pi$.

We will implement the $\text{len}$, $\text{contains}$, $\text{getitem}$ and $\text{iter}$ methods.

No $\text{setitem}$ or $\text{delitem}$: not appropriate.

iteration

Two possibilities:

- Class implements a $\text{getitem}$ method. Python can do iteration using that. Easy way and good strategy!
- We can do it ourselves. If we do, we can return the instance as the iterator, but we need to also provide a $\text{__next__}$ function.
iterators are objects

my_list = [1,2,3,4,5]
itr = my_list.__iter__()
itr # prints <listiterator object a 0x...>

iterators respond to next

my_list = [1,2,3]
itr = my_list.__iter__()
itr.__next__() # prints 1
itr.__next__() # prints 2
itr.__next__() # prints 3
itr.__next__() # StopIteration break
implementing __next__

- The value item holds the state of the present index for iteration.
- When the index goes past the end of the list, it raises an error.
- Python has special error names. It knows how to deal with StopIteration in an iterator.

Expansion iterator

```python
def __iter__(self):
    self.index=0
    return self
def __next__(self):
    if self.index < len(self.pi_list):
        val = self.pi_list[self.index]
        self.index += 1
        return val
    else:
        raise StopIteration
```
Expansion iterator: alternative

Use the list iterator
which uses list’s ‘next’ (so another isn’t needed)

def __iter__(self):
    self.index=0
    return self.pi_list.__iter__()
It doesn’t take long before these fractions have really big numbers. To see how well the sequence does, convert the sum to a float. Use the `__float__` method in `Rational`.

slices

The ‘modern’ approach to Python is to deal with slices in the `getitem` or `setitem` method.
is measures whether two variables refer to exactly the same object (not that the objects have the same value)

Only useful for mutables, results are "implementation dependent" on immutables

>>> L1 = [1,2,3]
>>> L2 = [1,2,3]
>>> L1==L2, L1 is L2
(True, False)

>>> L1 = L2
>>> L1==L2, L1 is L2
(True, True)
Not a copy

```python
>>> A = [1,2,3]
>>> B = A
>>> A == B, A is B
(True, True)
```

```python
>>> A
[1, 2, 5]
```

Copy: shallow

```python
>>> A = [1,2,3]
>>> B=A[:]
#or B=list(A)
>>> A==B, A is B
(True, False)
```

```python
>>> A
[1, 2, 3]
>>> B
[1, 2, 5]
```
Var name is A
Var value is arrow

```python
>>> A = [1, 2, 3]
>>> L = ['x', A, 'y']
>>> L
['x', [1, 2, 3], 'y']
>>> M = L[:]
# or M=list(L)
>>> L == M,
L is M
(True, False)
```

```python
>>> M[0]= 'a'
>>> M
['a', [1, 2, 3], 'y']
>>> L
['x', [1, 2, 3], 'y']
>>> M
['a', [1, 55, 3], 'y']
>>> L
['x', [1, 55, 3], 'y']
```
Copy: deep

```python
> import copy
> A = [1,2,3]
> L = ['x',A,'y']
> M = copy.deepcopy(L)
(True, False)
> M
['x', [1, 2, 3], 'y']
> L
['x', [1, 55, 3], 'y']
```

Note that the list embedded in M no longer has a name.

Copy for basic types

```python
>>> x = 'abc'
>>> y = x
>>> y == x, x is y
(True, True)
>>> x = x.replace('b','z')
>>> y == x, x is y
(False, False)
>>> x, y
('azc', 'abc')
>>> x = x.replace('z','b')
>>> y == x, x is y
(True, False)
```