What is a namespace?

- Namespace: mapping of names to values
- It looks like a dictionary, and for the most part it is (at least for modules and classes)
- Names in different namespaces have no relationship whatsoever

<table>
<thead>
<tr>
<th>main namespace</th>
<th>fun1 namespace</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 5</td>
<td>a “hi”</td>
</tr>
<tr>
<td>b False</td>
<td>x 1</td>
</tr>
<tr>
<td></td>
<td>b 0.0</td>
</tr>
</tbody>
</table>

Michigan State University
CSE 231, Summer 2013
Scope

- Textual region of a program where a namespace is directly accessible.
- A name is “in scope” if its namespace is accessible from the current location.

Multiple scopes

- Often, there can be multiple scopes that are candidates for determining a reference.
- Knowing which scope is the right one is important.
  (i.e. knowing the order of scope)
Two naming styles

- Unqualified (we’ve mostly been using this)
  - Variables: a = 10
  - Functions: my_func(50)

- Qualified (we’ve seen a few of these)
  - Modules: string.punctuation
  - Classes: clock.seconds = 30

For unqualified names, use **LEGEB**

- **Local**: inside the current function
- **Enclosing**: in an outer function
- **Global**: the module’s global names
- **Built-in**: Python’s standard library
For unqualified names: **LEGB Rule**

- **Local**: inside the current function

```python
def myFun(y):
    x = 1
    print(x, y)
    x = 5
    myFun(x+2)
    print(x)
    print(y)
```

---

For unqualified names: **LEGB Rule**

- **Local**: inside the current function

```python
def myFun(y):
    x = 1
    print(x, y)
    x = 5
    myFun(x+2)
    print(x)
    print(y)
```

---

Michigan State University  
CSE 231, Summer 2013
For unqualified names: **LEGB** Rule

- **Local:** inside the current function

```python
def myFun(y):
    x = 1
    print(x, y)
    x = 5
    myFun(x+2)
    print(x)
    print(y)
```

**top-level namespace**

`myFun:`

`x: 5`

**myFun's namespace**

`y: 7`
For unqualified names: **LEGB Rule**

- **Local**: inside the current function

```python
def myFun(y):
    x = 1
    print(x, y)
    x = 5
    myFun(x+2)
    print(x)
    print(y)
```

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-level</td>
<td></td>
</tr>
<tr>
<td>myFun:...</td>
<td>x: 5</td>
</tr>
<tr>
<td>myFun's</td>
<td></td>
</tr>
<tr>
<td></td>
<td>y: 7</td>
</tr>
<tr>
<td></td>
<td>x: 1</td>
</tr>
</tbody>
</table>

Prints: 1 7
For unqualified names: **LEGB Rule**

- **Local**: inside the current function

  ```python
  def myFun(y):
      x = 1
      print(x)
      x = 5
      myFun(x+2)
      print(x)
  print(y)
  ```

  Prints:
  1 7
  5

For unqualified names: **LEGB Rule**

- **Local**: inside the current function

  ```python
  def myFun(y):
      x = 1
      print(x)
      x = 5
      myFun(x+2)
      print(x)
  print(y)
  ```

  Prints:
  1
  5
  ERROR
For unqualified names: **LEGB** Rule

- **Enclosing**: an enclosing function definition

```python
def aFunc():
    x = 5
    def bFunc():
        print(x)
    bFunc()

aFunc()
bFunc()
```

---

**Enclosing**: an enclosing function definition

```python
def aFunc():
    x = 5
    def bFunc():
        print(x)
    bFunc()

aFunc()
bFunc()
```
For unqualified names: **LEGB Rule**

- **Enclosing: an enclosing function definition**

```python
def aFunc():
    x = 5
    def bFunc():
        print(x)
        bFunc()

aFunc()
bFunc()
```

**For unqualified names:**

- **LEGB Rule**

```python
def aFunc():
    x = 5
    def bFunc():
        print(x)
        bFunc()

aFunc()
bFunc()
```
For unqualified names: **LEGB Rule**

- **Enclosing**: an enclosing function definition
  ```python
def aFunc():
    x = 5
    def bFunc():
        print(x)
    bFunc()
  
  aFunc()
  bFunc()
```

- For unqualified names:
  - LEGB Rule
  - **Enclosing**: an enclosing function definition
    ```python
def aFunc():
    x = 5
    def bFunc():
        print(x)
    bFunc()
  
  aFunc()
  bFunc()
```

- Using a "global" variable (Bad Practice)
For unqualified names, use **LEGB**

- **Enclosing**: an enclosing function definition
  ```python
def aFunc():
    x = 5
    def bFunc():
        print(x)
    bFunc()

  aFunc()
  bFunc()
  ```

Prints: 5

---

For unqualified names: **LEGB** Rule

- **Enclosing**: an enclosing function definition
  ```python
def aFunc():
    x = 5
    def bFunc():
        print(x)
    bFunc()

  aFunc()
  bFunc()
  ```

Prints: 5
For unqualified names: **LEGB Rule**

- **Enclosing:** an enclosing function definition

  ```python
  def aFunc():
      x = 5
  def bFunc():
      print(x)
  bFunc()
  aFunc()
  ```

  Prints: 5
  ERROR

- **Global:** the global namespace

  ```python
  x = 1
  def myFun():
      print x
  myFun()
  ```

  top-level namespace
For unqualified names: **LEGB Rule**

- **Global:** the global namespace

```python
x = 1

def myFun():
    print x

top-level namespace
x : 1
myFun:...
```

```
myFun()
```

---

For unqualified names: **LEGB Rule**

- **Global:** the global namespace

```python
x = 1

def myFun():
    print x

myFun()  
```

---

Using a global variable (Bad Practice)
For unqualified names: **LEGB Rule**

- **Global:** the global namespace

  ```python
  x = 1
  def myFun():
    print x
  ```

  myFun()

  Prints: 1

- **Built-in:** Python’s standard library

  ```python
  print(abs(-5))  # prints 5
  print(len("asdf"))  # prints 4
  dir(__builtins__)  # check it out!
  ```
Unqualified names: **LEGB Rule**

- **Local**, defined in the current function?
- *if not there*, **Enclosing**: defined in an enclosing function?
- *if not there*, **Global**: defined in the main module?
- *finally*, **Built-in**: defined in the built-in scope?
- *else* ERROR

Two naming styles

- **Unqualified** (we’ve mostly been using this)
  - Variables: \( a = 10 \)
  - Functions: \( \text{myFunc}(50) \)

- **Qualified** (we’ve seen a few of these)
  - Modules: \( \text{string.punctuation} \)
  - Classes: \( \text{clock.seconds} = 30 \)
Qualified namespace scope

- **Recall**
  - Classes: clock.seconds = 30
  - Modules: string.punctuation

- Qualified namespace scopes follow **LCB**

---

The **LCB** rule for qualified names

- **Local:** look inside the local object

```python
class MyClass(object):
    pass

my_inst = MyClass()
my_inst.my_var = 12
print(my_inst.my_var)
```
The **LCB** rule for qualified names

- **Local**: look inside the local object
- **Class**: look inside the class definition

```python
class MyClass(object):
    my_var = 100

my_inst = MyClass()
print(my_inst.my_var)
```

- **Base**: look inside the base classes
Unlike some OOP systems…

- In Python, classes are also objects. You can assign local values to a class:
  ```python
class MyClass(object):
    pass
MyClass.x = 5
print(MyClass.x)  # prints 5
```

- In Python, once an instance is created, it can be modified in any way.
- This can be confusing (and dangerous!)

qualified follows the LCB rule

- **Local:** look inside the local object first
- **Class:** if not in the object, look in the class of the object
- If not in either of those, look in any classes the object’s **Base** class (inheritance)
- else ERROR
Tricky stuff

```python
class MyClass(object):
    x = 100

my_inst = MyClass()
print(my_inst.x)
MyClass.x = 500
print(my_inst.x)
```

This is why we have constructors!

Not-so-tricky stuff

```python
class MyClass(object):
    def __init__(self):
        self.x = 100

my_inst = MyClass()
print(my_inst.x)
MyClass.x = 500
print(my_inst.x)
```

Constructors ensure that instances have the correct local variables attached to them.
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)
```

Private variables in an instance

- Many OOP approaches allow you to make “private” an instance variable or function.
- Private means not accessible outside of the class definition.

```python
class MyClass(object):
    def fun(self):
        pass

# can access fun here
# cannot access fun here
```
Privacy in Python

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

```python
__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
```

Different about Python

1. Unlike some OOP systems, classes are also objects in the system. You can assign local values to a class.
2. Unlike some OOP systems, once an instance is made it can be modified in any way a user chooses.

This leads to some interesting effects.
Local in the instance

Assignment to the instance makes the variable local.

```python
class MyClass(object):
    pass
my_inst = MyClass()
my_inst.my_var = 12
print(my_inst.my_var)  # prints 12
```

Look in the class

```python
class MyClass(object):
    my_var = 100
my_inst = MyClass()
print(my_inst.my_var)  # prints 100
```
Local assignment before class
This mix can lead to some tricky stuff.

```python
class MyClass(object):
    my_var = 100

my_inst = MyClass()
your_inst = MyClass()
my_inst.my_var = -1
print(my_inst.my_var)  # prints -1
print(your_inst.my_var) # prints 100
MyClass.my_var = 500
print(my_inst.my_var)  # prints -1
print(your_inst.my_var) # prints 500
```

That’s why the constructor

- If you create a constructor, you assure that you locally create the variables you want to show up in the initial instance made from each class.
- ‘global’ variables in the class are not really global.
qualified namespaces are dicts

- the namespaces in every object and module is indeed a namespace
- that dictionary is bound to the special variable `__dict__`
- it lists all the local vals (variables, functions) in the object

example namespace

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

>>> MyClass.__dict__.items()
[('__module__', '__main__'), ('my_method', <function my_method at 0x12200f0>),('__dict__', <attribute '__dict__' of 'MyClass' objects>), ('x', 100), ('__weakref__', <attribute '__weakref__' of 'MyClass' objects>), ('__doc__', None)]
private variables in an instance

- many OOP approaches allow you to make a variable or function in an instance private
- private means not accessible by the class user, only the class developer.
- there are advantages to controlling who can access the instance values

‘privacy’ in Python

- Python takes the approach: “We are all adults here”. No hard restrictions.
- Provides naming to avoid accidents. Use __ in front of any variable.
- This ‘mangles’ the name to include the class, namely __var becomes _class__var
- Still fully accessible, and the __dict__ makes it obvious
```python
class MyClass(object):
    def __init__(self, p1='firstParam', p2='secondParam'):
        self.var1 = p1
        self.__var2 = p2

my_inst = MyClass()
my_inst.__dict__.items()
[['_MyClass__var2', 'secondParam'], ('var1', 'firstParam')]
```

Example