Python Intermediate Programming

Unit 0: Introduction

PYTHON REVIEW A: VARIABLES, THEIR SCOPE, AND BINDING

DR. ERIC CHOU IEEE SENIOR MEMBER

Objectives

- Review Python Programming Essentials and Python Object-Oriented Programming topics

Overview of Python Language

LECTURE 1

Python in Four Sentences:

1. Names (in namespaces) are bound to objects.
2. Everything that Python computes with is an object. (examples are instance/data, function, module, and class objects)
3. Every object has its own namespace. (a dictionary that binds its internal names to other objects)
4. Python has rules about how things work.

Names (in namespaces) are bound to objects.

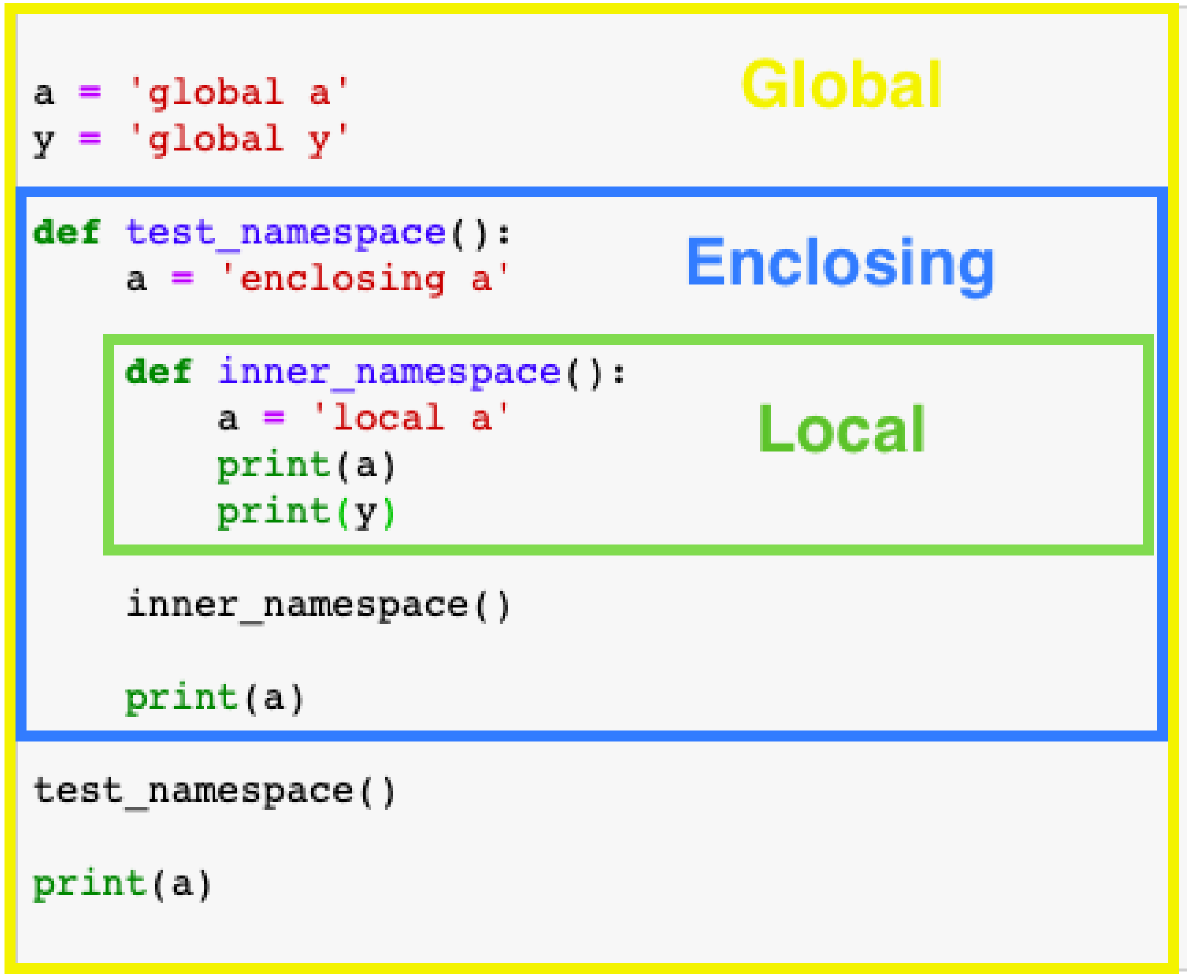
- Every name appears in the namespace of some object (when we define names in a module, for example, these names appear in the module object's namespace); and every name is itself bound to some object (names are bound when defined on the left of the = symbol; they can be rebound on the left using the = symbol again; names are also bound in import statements, discussed later).

Namespaces and modules

A **namespace** is a mapping from names to objects

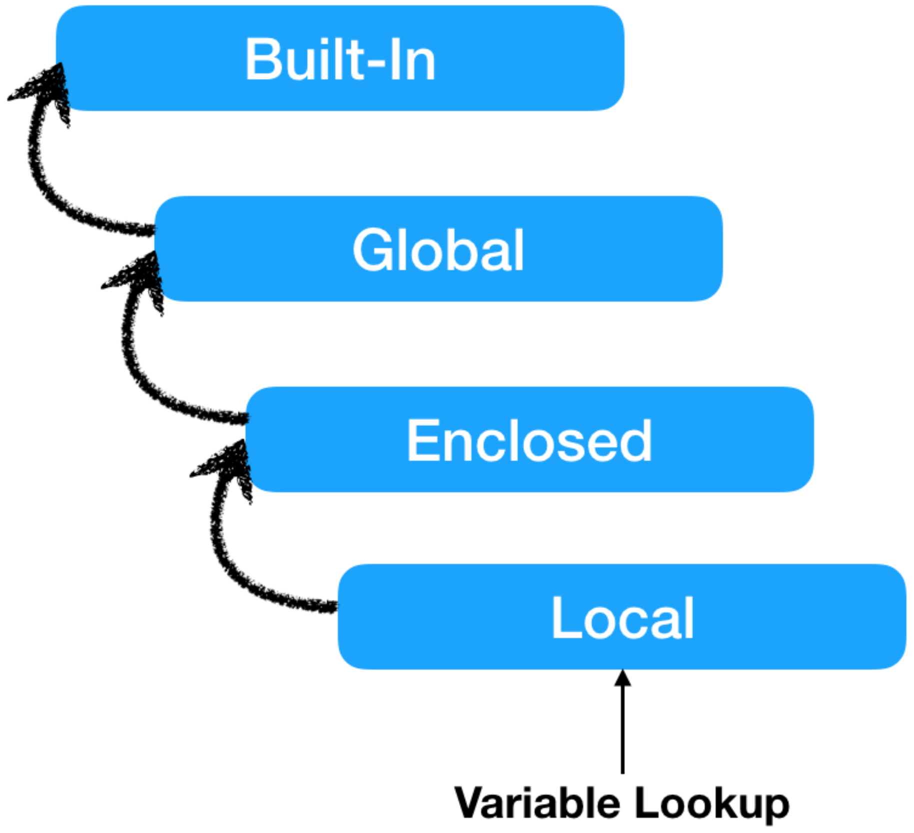
Several different kinds of namespaces:

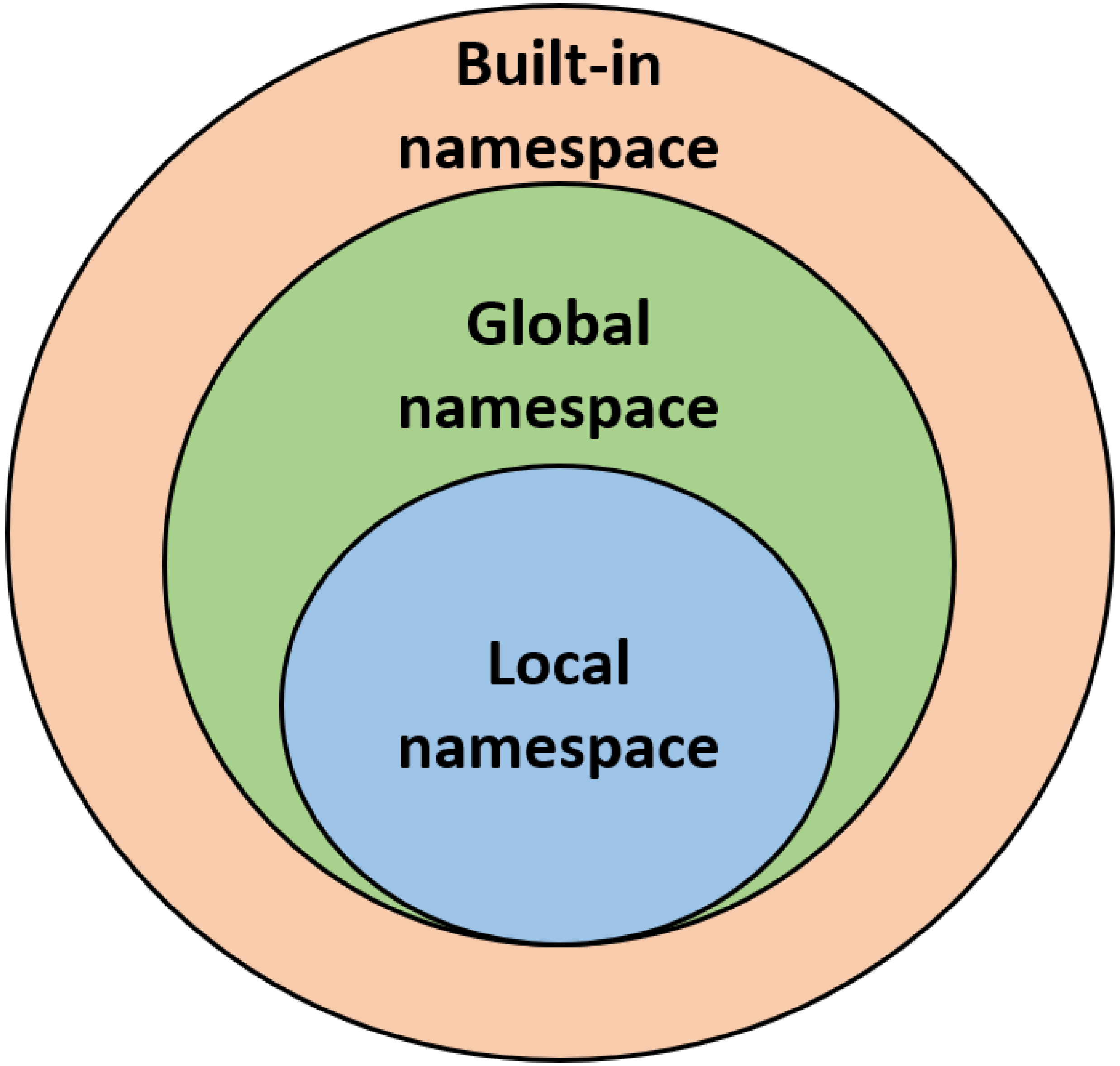
- Global: associated with idle window
- Local: associated with a function
- Module: associated with a .py file that is imported into another namespace



```
a = 'global a'
y = 'global y'

def test_namespace():
    a = 'enclosing a'

    def inner_namespace():
        a = 'local a'
        print(a)
        print(y)

    inner_namespace()

    print(a)

test_namespace()

print(a)
```

```
local a
global y
enclosing a
global a
```

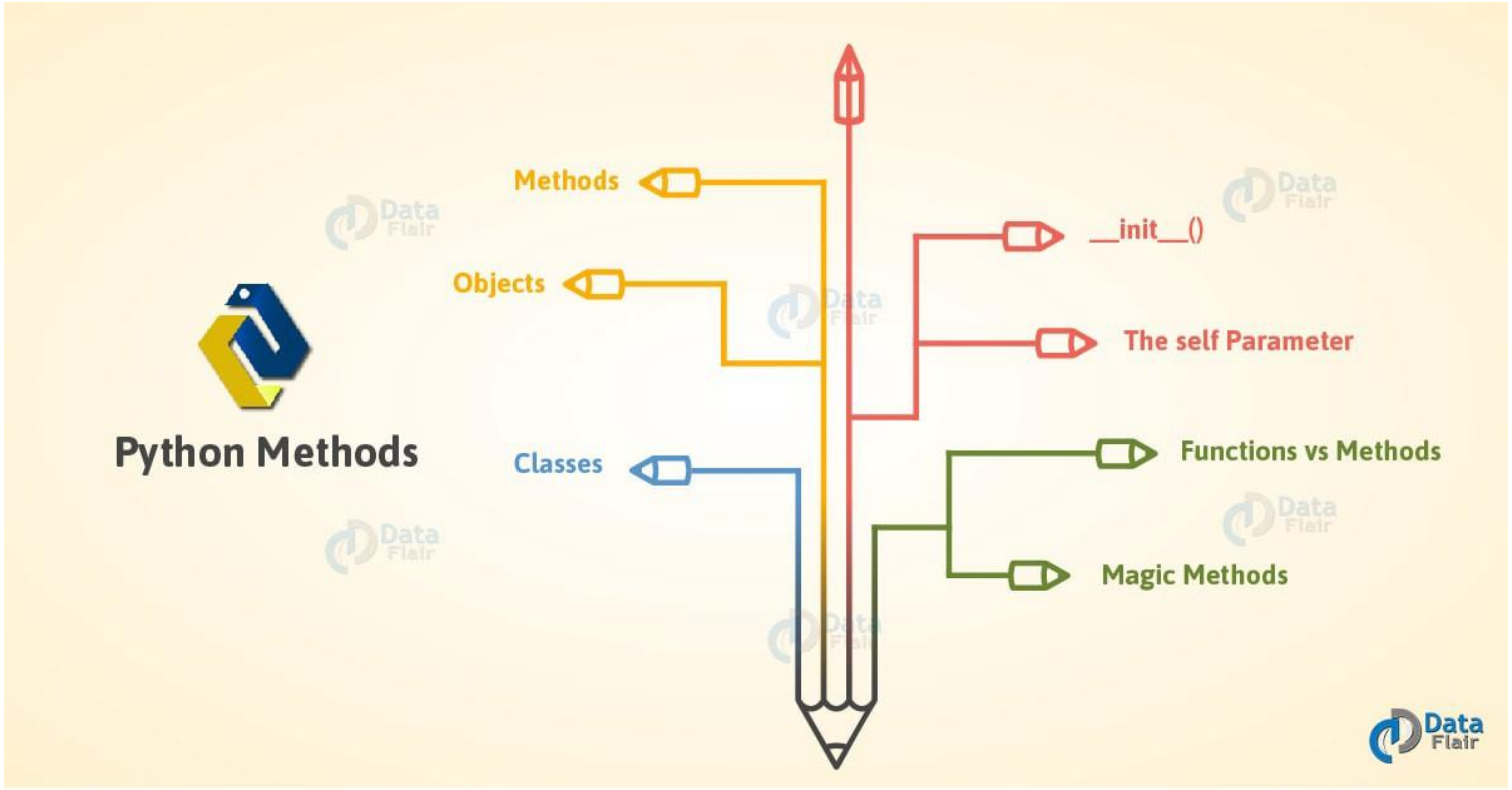




Type of Namespaces

Objects are the fundamental unit with which Python computes

1. We can compute with int objects (instance objects from the int class) by using operators; for the int object bound to x by x = 1 we can rebind it to another int object, one bigger, by writing x = x + 1. We will learn later that Python translate x+1 into the method call x.__add__(1)
2. We can compute with function objects by calling them; for the function object bound to print, we can write print(x); we will learn later in this lecture that we can also pass functions as arguments to functions and return functions as results from functions.


Python Methods
Methods
Objects
Classes
__init__()
The self Parameter
Functions vs Methods
Magic Methods
Data Flair

Objects are the fundamental unit with which Python computes

3. We can compute with module objects by importing them (and/or the objects bound to the names in their namespaces).

```
import random

x = random.randint(1,6)
```

The name random is bound to the `random` module object

```
from random import randint

x = randint(1,6)
```

The name `randint` is bound to the `randint` function object defined in the random module.

4. We can compute with class objects by constructing instance objects and using the instances to call class methods. For example, we can write

```
    timer = Stopwatch()
    timer.start()
```

Binding

LECTURE 2

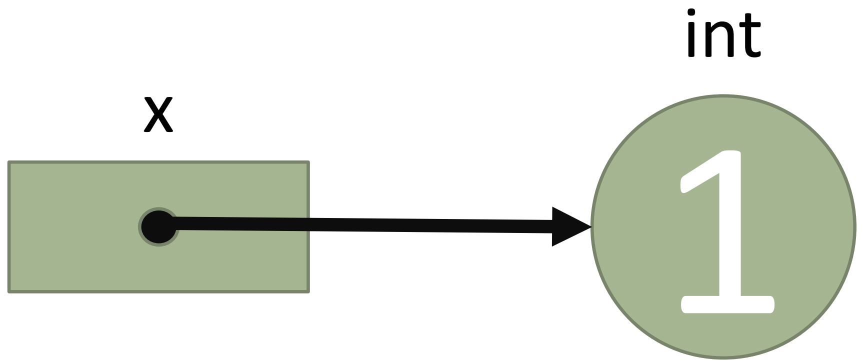
Binding (and Drawing Names and their associated Objects)

- The process of making a name refer to a value: e.g., x = 1 binds the name x to the value 1 (which is an object/instance of the int class); later we can bind x to another value (possibly from another class) in another assignment: e.g., x = 'abc'. We speak about "the binding (noun) of a name" to mean the value (such values are always objects) that the name is currently associated with (or the object the name refers to).

- In Python, every data instance, module, function, and class is an object that has a dictionary that stores its namespace: all its internal bindings. We will learn much more about namespaces (and how to manipulate them) later in the quarter, when we study classes in more detail.

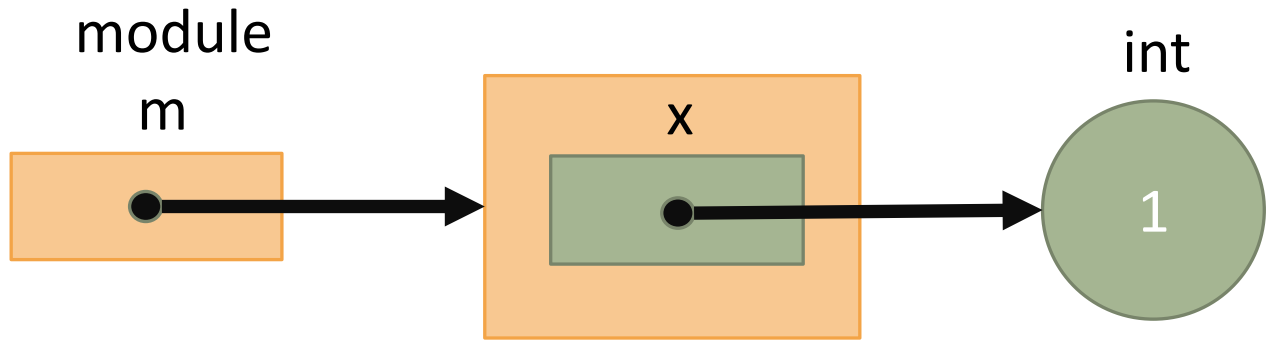
Binding

- Typically, we illustrate the binding of a name to an object (below, x = 1) as follows. We write the name over a rectangle, in which the tail of an arrow is INSIDE, specifying the current reference stored for that name: the arrow's head refers to a rounded-edge rectangle object labeled by its type (here the class it is constructed from) and its value (inside).



Binding

- Technically, if we we write x = 1 inside the module m, Python has already created an object for module m (we show all objects as rounded-edge rectangles) and it puts x, its box, and its binding in the namespace of module m: here, the name x is defined inside module m and bound to object 1. That is, we would more formally write the result of x = 1 in module m as



- But often we revert back to the previous diagram, when we don't care what module in which x is defined.

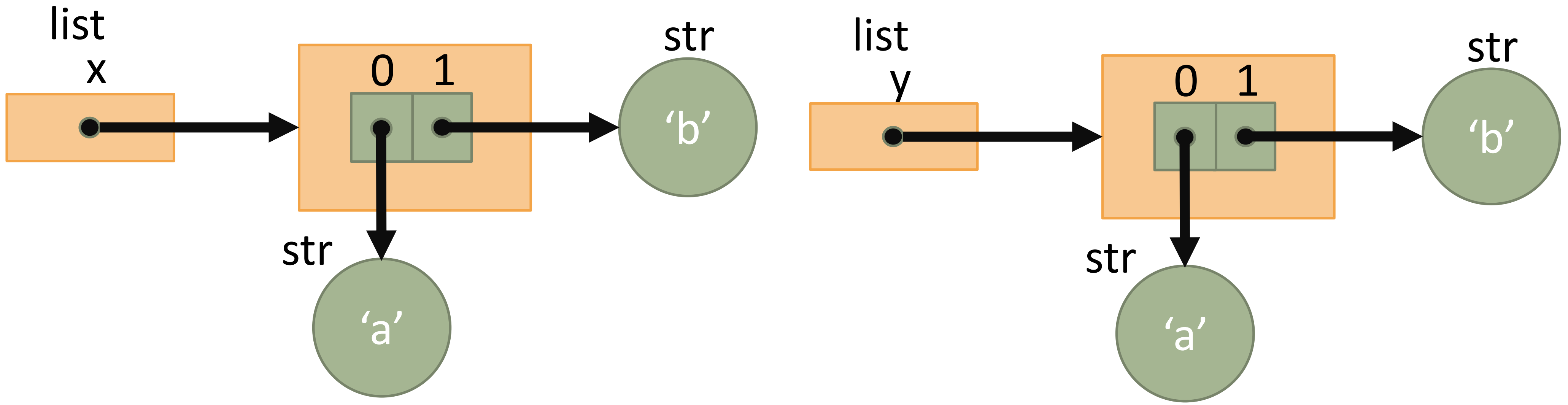
Binding

- Finally, the "is" operator in Python determines whether two references refer to the same object; the `==` operator determines whether two references refer to objects that store the same internal state. If `a` is `b` is True then `a == b` must be True (because every object has the same state as itself).
- If we write

```
x = ['a','b']
y = ['a','b']
```

Is and ==

- then x is y is False and x == y is True: the names x and y are bound to/refer to two different list objects (each use of [] creates a new list), but these two objects store the same state ('a' at index 0; 'b' at index 1).



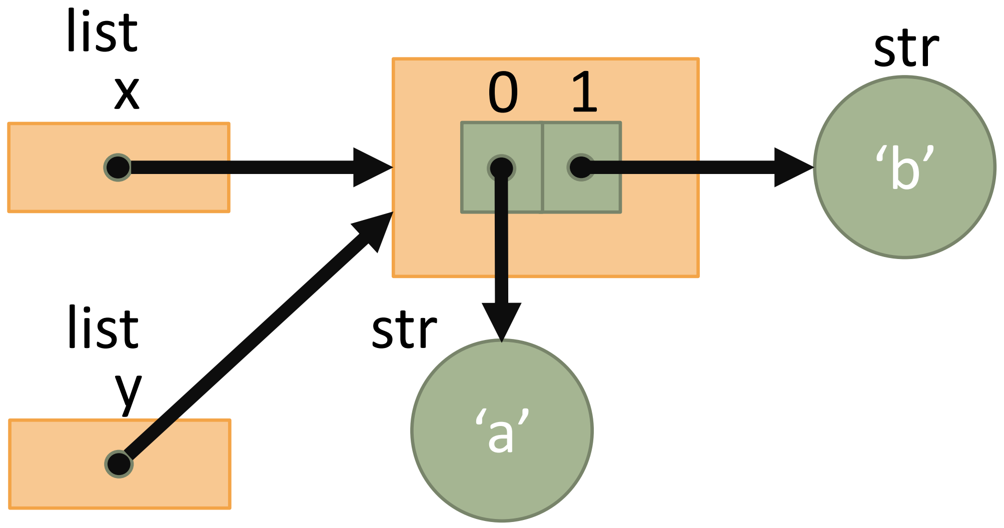
Is and ==

- Likewise, if we write

```
x = ['a','b']
y = x
```

then we create one object, and bind it to both x and y (assignment y = x just copies into name y the reference in name x, making y and x refer to the same object). Here, x is y is True (and x == y is therefore True): the names x and y refer to the same list object. We would diagram it as You should be able to draw simple picture of these names and objects (both the list and int objects) to illustrate the difference between the "is" and == operators.

- What happens in each example (what picture results) if we execute y[0] = 'c'?



Statements and Expressions

LECTURE 3

Statements and Expressions

- **Statement** are executed to cause an effect (e.g., binding/rebinding a name or producing output).
- **Expressions** are evaluated to compute a result (e.g., computing some formula, numeric, string, boolean, etc.).
- For example, the statement x = 1, when executed, causes a binding of the name x to an object representing the integer 1. The expression x+1, when evaluated, computes the object representing the integer 2. Typically, we write expressions inside statements: two examples are x = x+1 and print(x+1): both "do something" with the computed value x+1 (the first binds x to it; the second prints it). The distinction between statements and expressions is important in most programming languages. Learn the technical meaning of these terms.

Control Structures

- Control structures (even simple sequential ones, like blocks) are considered to be statements in Python (and Python has rules describing how to execute them). Control structures might contain both statements and expressions. The following is a conditional statement using if/else

```
if x == None:
    y = 0
else:
    y = 1
```

- This if statement contains the expression x == None and the statements y = 0 and y = 1. Technically, x, None, 0 and 1 are "trivial" expressions (which trivially compute the object to which x is currently bound, and the object values None, 0, and 1 which are literals: preconstructed objects).

Control Structures

- The following statement includes a conditional expression that binds a value to y: the conditional expression includes the expressions 0 (yes, an object by itself, or just a name refering to an object, is a simple expression), x == None, and 1.

```
    y = (0 if x == None else 1)
```

- We will discuss conditional statements vs. conditional expressions in more detail later in this lecture note.

None

- **None** is a value (object/instance) of NoneType it is the only value of that type. Sometimes we use None as a default value for a parameter's argument; sometimes we use it as a return value of a function: in fact, a Python function that terminates without executing a return statement automatically returns the value None.
- If None shows up as an unexpected result printed in your code or more likely in a raised exception, look for some function whose return value you forgot to specify explicitly (or whose return statement somehow didn't get executed before Python executes the last statement in a function).

pass

- **pass** is the simplest statement in Python; semantically, it means "do nothing". Sometimes when we write statements, we aren't sure exactly which ones to write, so we might use pass as a placeholder until we write the actual statement we need. Often, in tiny examples, we use pass as the meaning of a function.

```
def f(x) : pass       or    def f(x) :
                                pass
```

- If you are ever tempted to write the first if statement below, don't; instead write the second one, which is simpler: but, they produce equivalent results.

```
 if a in s:                    # DON'T write this code
     pass
 else:
     print('a is not in s’)
if a not in s:                 # Write this code instead; it is equivalent
     print('a is not in s’)    # and simpler; strive to use Python simply
```

Importing

- There are five import forms; you should know what each does, and start to think about which is appropriate to use in any given situation. Note that in EBNF (a notation used to describe programming languages, which we will discuss soon) [...] means option and {...} means repeat 0 or more times, although this second form is sometimes written (...)* when describing the syntax of Python.
- Fundamentally, import statements bind names to objects (one or both of which come from the imported module).

Ways of Importing Module or Classes

"import module-name" form:

1. import module-name{,module-name}
2. import module-name [as alt-name] {,module-name [as alt-name]}

"from module-name import" form:

3. from module-name import attr-name{,attr-name}
4. from module-name import attr-name [as alt-name] {,attr-name [as alt-name]}
5. from module-name import *

attr-name

- Above, **alt-name** is an alternative name to be bound to the imported object; **attr-name** is an attribute name already defined in the namespace of the module being imported.
- The "import module-name" forms import the names of modules (not their attribute names). (1) bind each module-name to the object representing that imported module-name. (2) bind each **alt-name** to the object representing its preceding imported **module-name**. Using a module name, we can refer to its attributes using the **module-name.attribute-name** form.

Importing

- The "from module-name import" forms don't import a module-name, but instead import some/all attribute names defined/bound inside module-name. (3) bind each attr-name to the object bound to that attr-name in module-name. (4) bind each alt-name to the object bound to the preceding attr-name in module-name. (5) bind each name that is bound in module-name to the same object it is bound to in module-name.
- Import (like an assignment, and a def and class definition) creates a name (if that name is not already in the namespace) and binds it to an object: the "import module-name" form binds names to module objects; the "from module-name import" form binds names to objects (instances, functions, modules, classes, etc.) defined inside modules (so now two modules contain names - maybe the same, maybe different, depending on which of form 3 or 4 is use-bound to the same objects).

Importing

- The key idea as to which kind of import to use is to make it easy to refer to a name but not pollute the name space of the module doing the importing with too many names (which might conflict with other names in that module).
- If a lot of different names in the imported module are to be used, or we want to make it clear when an attribute name in another module is used, use the "import module-name" form (1) and then qualify the names when used: for example

```
import random
#use:random.choice(...) and random.randint(...)
```

Importing

- If the imported module-name is too large for using repeatedly, use an abbreviaton by importing using an alt-name (2) : for example

```
import high_precision_math as hp_math
# use:  hp_math.sqrt(...)
```

- If only one name (or a few names) are to be used from a module, use the form (3):

```
from random import choice, randint
# use: choice(...)  and  randint(...)
```

Importing

- Again, use alt-name to simplify either form, if the name is too large and unwieldy to use. Such names are often very long to be clear, but awkward to use many times at that length. Generally we should apply the Goldilocks principle: name lengths shouldn't be too short (devoid of meaning) or too long (awkward to read/type) but their length should be "just right". Better to make them too long, because there are ways (such as alt-name) to abbreviate them.
- We almost never write the * form of importing. It imports all the names defined in module-name, and "pollutes" our namespace by defining all sorts of names we may or may not use (and which might conflict and redefine names that we have already defined). Better to explicitly import the names needed/used. Eclipse marks with a warning any names that are imported but unused.

Iteration/Range

LECTURE 4

Iteration/Range

- Directly iterating over values in a list vs. Using a range to iterate over indexes of values in a list
- We know that we can iterate (with a for loop) over ranges: e.g., if alist is a list, we can write

```
alist = [5, 2, 3, 1, 4, 0]
for i in range(0,len(alist)): # like for i in range(len(alist)):
    print(alist[i])
```

- Here i takes on the values 0, 1, ... , len(alist)-1 but not len(alist), which is 6. The code above prints all six values in alist: alist[0], alist[1], ... alist[5].

Iteration/Range

- Often, we want to iterate over all the values in a list (alist) but don't need to know/use their indexes at all: e.g., to print all the values in a list we can use the loop

```
for x in alist:
    print(x)
```

which is much better (simpler/clearer) than the loop

```
for i in range(len(alist)):
    print(alist[i])
```

Iteration/Range

- although both produce the same result. Generally, choose to write the simplest loop possible for all your code. Sometimes you might write a loop correctly, but then realize that you can also write a simpler loop correctly: change your code to the simpler loop. Sometimes (when doing more complicated index processing) we must iterate over indexes. Use the simplest tool needed to get the job done.
- In many cases where using range is appropriate, we want to go from the low to high value inclusively. I have written function named irange (for inclusive range) that we can import from the goody module and use like range.

Iteration/Range

```
from goody import irange
for i in irange(1,10):
    print(i)
```

prints the values 1 through 10 inclusive; **`range(1,10)`** would print only 1 through 9. One goal for ICS-33 is to show you how to write alternatives to built-in Python features; we will study how **`irange`** is written later in quarter, but you can import and use it now (and examine its definition in the goody module).

I have also written **`frange`** in goody, which allows iteration over floating point (not int) values.

Iteration/Range

```
from good import frange
for x in frange(0., 1., .5):
    print(x)
```

prints (iterating from 0 to 1 in steps of .5

```
0.0
0.5
1.0
```

Arguments and Parameters

LECTURE 5

Arguments and Parameters

- Whenever we DEFINE a function (and define methods in classes), we specify the names of its parameters in its header (in parentheses, separated by commas).

- Whenever we CALL a **`function`** we specify the values of its arguments (also in parentheses, separated by commas). The definition below

```
 def f(x,y):
     return x**y
```

defines a function of two parameters: x and y. When we define a function we (re)bind the function's name to the function's object. This is similar to what happens when we write x = 1 (which binds a name to a data object: an instance of the int class).

Arguments and Parameters

- Calling `f(5,2*6)` calls this function with two arguments: the arguments 5 and 2*6 are evaluated (producing objects) and the values/objects computed from these arguments (5 and 12) are bound to their matching parameters in the function header (and then the body of the function is executed). Function calls happen in two steps: bind the arguments to the parameters; then execute the body of the function using the parameter names. So, parameters are names inside the name-space of the function.
- We will discuss the details of argument/parameter binding in much more detail below, but in a simple example like this one, parameter/argument binding is like writing: x = 5 and then y = 2*6 which binds the parameter x to the object 5 and then the parameter y to the object 12.

Arguments and Parameters

- Sometimes we can use the parameter of a function as an argument to another function call inside its body. If we define

```
def factorial_sum(a,b):
    return factorial(a) + factorial(b)
```

- Here, the parameters a and b of `factorial_sum` function are used as arguments in the two calls in its body to the factorial function.

Arguments and Parameters

- Parameters are always names. Arguments are expressions that evaluate to objects. Very simple expression include literals (like 1 and 'abc') and any names bound to values (as a and b ar in the calls to factorial above).
- It is important that you understand the distinction between the technical term PARAMETER and ARGUMENT, and that calling a function first binds the parameter names to their associated argument values in the `function`'s HEADER (we will discuss the exact rules soon) and then executes the BODY of the function.

Function calls

LECTURE 6

Function calls ... always include ()

how we can pass a function as an argument to another function

- Any time a reference to an object is followed by (...) it means to perform a function call on that object (some objects will raise an exception if they do not support function calls: writing 3() or 'abc'() will both raise exceptions).
- While this rule is simple to understand for the functions that you have been writing and calling, there are some much more interesting ramifications of this simple rule. Run the following code to define these three functions.

Function calls … always include ()

how we can pass a function as an argument to another function

```
def double(x):
    return 2*x
def triple(x):
    return 3*x
def times10(x):
    return 10*x
```

- Note that each def defines the name of a function and binds that name to the function object that follows it.

Function calls ... always include ()

how we can pass a function as an argument to another function

- If we wrote

```
 f = double
```

then f would become a defined name that is bound to the same (function) object that the name double is bound to (like like writing y = x if x were bound to an integer). The expression "f is double" would evaluate to True, because these two names are bound to the same function object. Note that there are no () in the code above, so there is no function call.

Function calls … always include ()

how we can pass a function as an argument to another function

- If we then write

```
print( f(5) )
```

- Python would print 10, just as it would if we wrote

```
  print( double(5) )
```

because f and double refer to the same function object, and it makes no difference whether we call that function object using the name f or double. The function call does not occur until we use (). Of course, the () in print(...) means that the print function is also called: print's argument is the result returned by calling f(5). So, the two sets of () in th print(f(5)) means that Python calls two functions.

Function calls … always include ()

how we can pass a function as an argument to another function

- Here is a more interesting example, but using exactly the same idea.

```
for f in [double, triple, times10]:
    print(f(5))
```

- Here f is a variable that iterates over (is bound to) all the values in a list (we could also have used a tuple): each value in the list is a reference to a function object (the objects referred to by the names double, triple, and times10). This code in the loop's body prints the values computed by calling each of these function objects with the argument 5. Note that these functions are NOT called when creating the list (no parentheses there!): the list is just built to contain references to these three function objects: again, when their names are not followed by () there is no function call. This code prints 10, 15, and 50.

Function calls ... always include ()

how we can pass a function as an argument to another function

- Using the same logic, we could also write a dictionary whose keys are strings and whose values are function objects, and then use a string as a key needed to retrieve and call its associated function object.

```
fs = {'x2' : double, 'x3' : triple, 'x10' : times10}
print( fs['x3'](5) )
```

- Here fs is a **`dictionary`** that stores keys that are strings and associates each with a function object: there are no calls to functions when building the dictionary for fs in the first statement - no (); we then can retrieve the function associated with any key (here the key 'x3') and then call the resulting function object (here fs['x3']) with the argument 5. Of course, in the second statement we use the () to call the function selected from the dictionary.

Function calls

- We can also pass (uncalled) functions as arguments to other functions.

```
def count_true(f, alist):
    count = 0
    for x in alist:
        if f(x):
            count += 1
    return count
```

if we wrote

```
from predicate import is_prime
```

(which imports the is_prime function from the predicates module: a predicate is a function of one argument that returns a boolean) then calling count_true(is_prime, [2,3,4,5,6,7,8,9,10]) returns 4 (only the numbers 2, 3, 5, and 7, are prime.

Function calls

- Note that the parameter f is bound to the function object that **is_prime** is bound to (**is_prime** is not called when it is specified as an argument), and f is called many times inside the **count_tree** function: one for each value in the list. We call **count_true** a "functional" because it is passed a function as an argument.

Global and Local Names

LECTURE 7

Global and Local Names

- Names defined in a module are global definitions; names defined in a function (and later names defined in a class) are local definitions. In a module, we can refer to global names, but not any local names. In functions we can refer to local or global names.
- Let's start by looking at

```
x = 1
print(x)
```

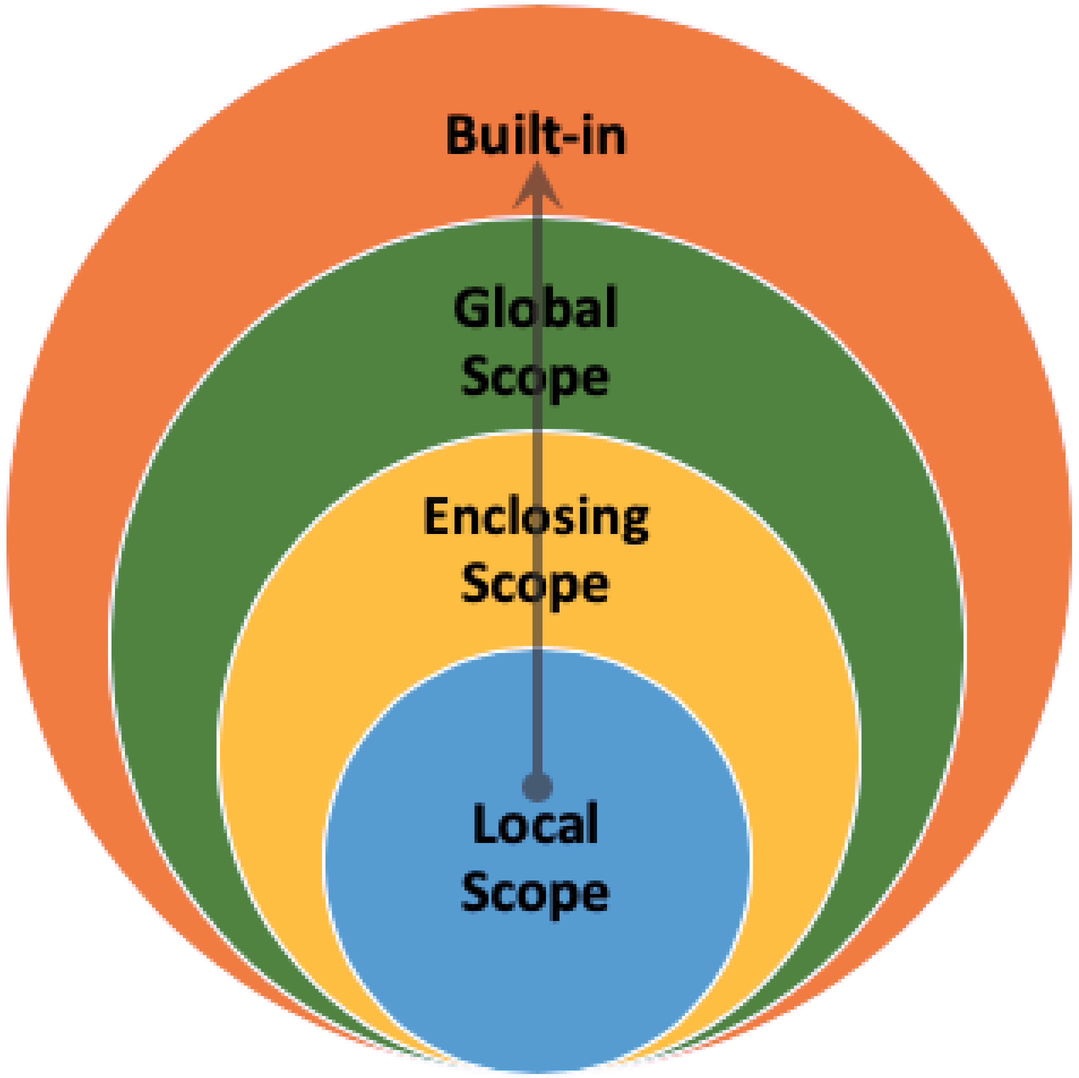
- which prints 1. Here the module defines a global name x and prints the value bound to that global name.

Global and Local Names

- Now let’s look at

```
x = 1
def f():
    print(x)
f()
print(x)
```

- which prints 1 and then 1. Here the module defines a global name x and a global name f (the function, which uses the global name x). When we call f(), it prints the value bound to the global name x. Then, returning to the module, it prints the value bound to the global name x again.


Built-in
Global Scope
Enclosing Scope
Local Scope

Global and Local Names

- Now let’s look at

```
x = 1
def f():
    x = 2
    print(x)
f()
print(x)
```

- which prints 2 and then 1. Here the module defines a global name x and a global name f (the function, which defines a local name x). When we call f(), it binds the local name x to 2 and then prints the value bound to the local name x. Then, returning to the module, it prints the value bound to the global name x.

Global and Local Names

- What is the primary difference between this example and the preceding one: When printing x in the function f, Python first looks for a local name x, if it finds one it uses its value; if it doesn't find a local name, it tries to find a global name and uses its value.

Global and Local Names

- Now let's look at

```
x = 1
def f():
    y = 2
    print(x,y)
f()
print(x)
```

which prints 1 2 and then 1. Here the module defines a global name x and a global name f (the function, which defines a local name y). When we call f(), it binds the local name y to 2 and then prints the value bound to the global name x and the local name y. Then, returning to the module, it prints the value bound to the global name x. So, this example illustrate the use of a global and local name.

Global and Local Names

- If we replaced **`print(x)`** at the end of the module with **`print(x,y)`** Python would raise a **`NameError`** exception because y is a local name defined only in the function f. It is not defined in the module.
- Finally, what if we wanted function f to not only print the original value of the global name x, but also to CHANGE it to 2. We CANNOT do it as follows, but the reason is a bit deep.

```
x = 1               # wrong code
def f():            # wrong code
    y = 2           # wrong code
    print(x,y)      # wrong code
    x = 2           # wrong code
f()                 # wrong code
print(x)            # wrong code
```

Global and Local Names

- What happens when Python executes this code? Here the module defines a global name x and a global name f (the function, which defines local names y and x).
- When we call f(), it binds the local name y to 2 and then it TRIES to print the value bound to the local name x and the local name y. But the local name x has no binding yet, so it raises the **`UnboundLocalError`** exception.

Global and Local Names

- Note that if we put the x = 2 before the print

```
x = 1               # wrong code
def f():            # wrong code
    y = 2           # wrong code
    x = 2           # wrong code
    print(x,y)      # wrong code
f()                 # wrong code
print(x)            # wrong code
```

- Python would print 2 2 and then print 1, executing without raising an exception but still not solving our problem. It still defines x as a local name and binds it to 2, now allowing the print to work, but still leaving the global name x unchanged.

Global and Local Names

- We can change the value in the global name x by using a "global" declaration inside f.

```
x = 1
def f():
    global x
    y = 2
    print(x,y)
    x = 2
f()
print(x)
```

Global and Local Names

- Python would print 1 2 and then print 2 (thus changing the global name x). Here the module defines a global name x and a global name f (the function, which defines only a local name y; the global x declaration tells Python to use the global name x inside f whenever x is used). When we call f(), it binds the local name y to 2 and then it prints the value bound to the global name x and the local name y. Then it changes the binding of the global name x to 2 (recall because of the global x declaration, all uses of x inside f refer to the global name x). Then, returning to the module, it prints the value bound to the global name x, which is now 2.

Global and Local Names

- In summary, we can always use a global name inside a function to find its value without doing anything special, but if we want to rebind the global name to a new value inside the function we must declare the name global in the function.
- What do you think the following code will do? Notice that is similar to the code above, but it (a) does not define a global name x before defining f and (b) omits the print statement.

```
def f():
    global x
    y = 2
    x = 2
f()
print(x)
```

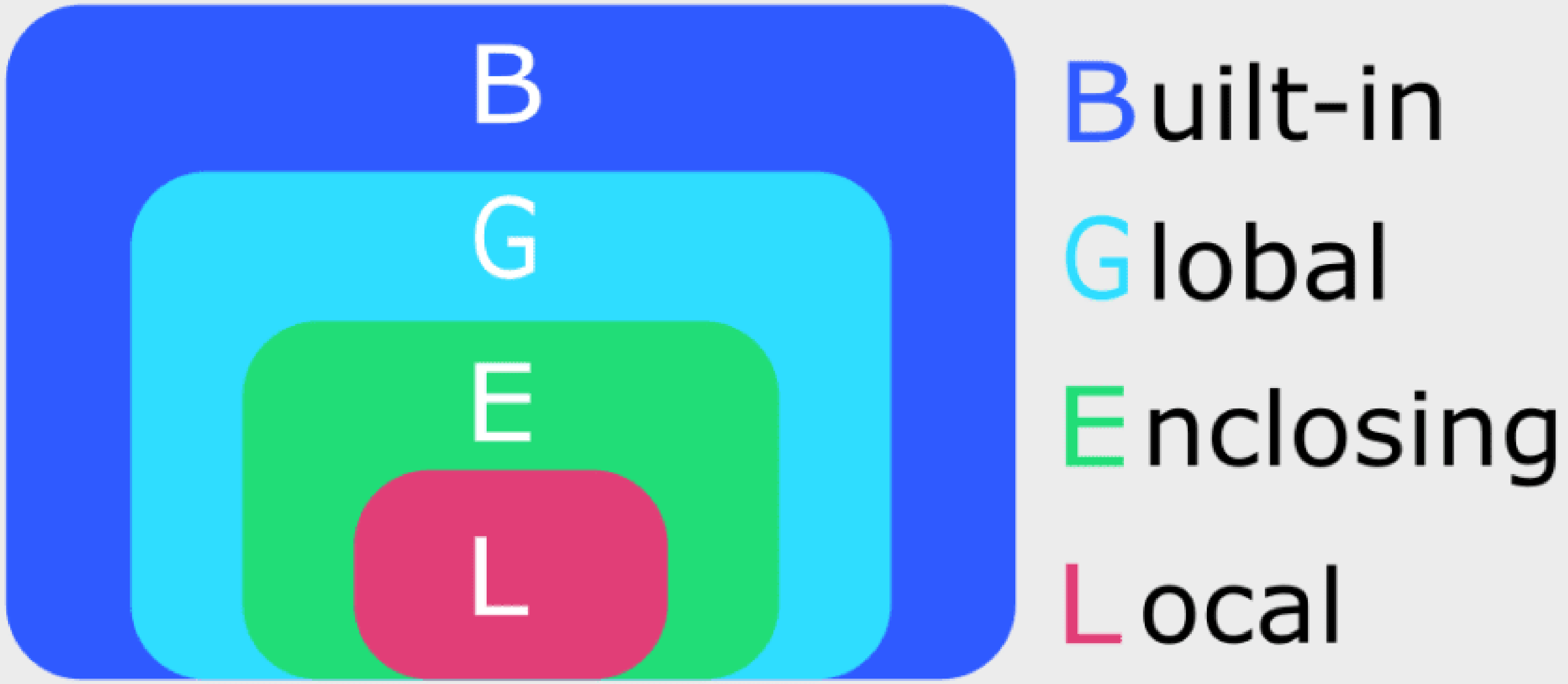
Global and Local Names

- Also, can you explain what would happen if the print statement were restored, between the statements x = 2 and y = 2.
- Bottom Line: You should know how global names are found/used inside functions, although typically no global names should be used: all information that a function uses should be passed into the function via its parameters.

LEGB rule for bindings

LECTURE 8

Scopes in Python



LEGB rule for bindings

- Look at the following function named bigger_than, which is a function of one parameter. Inside, it defines a local function named test which also has one parameter, then the bigger_than function returns a reference to the test function object that it defines. Have you seen functions that return functions before? This is powerful programming feature.

```
def bigger_than(v) :
    def test(x) :
        return x > v
    return test
```

LEGB rule for bindings

- Note that the inner function (test) can refer to global names (defined in the module) and any local names defined in the outer function (bigger_than): here test refers to its own local parameter name (x) and to the parameter name (v) defined if the bigger_than function. Generally, when Python looks up the binding of any name, it uses the LEGB rule, finding the object bound to the name using the following ordering.

  L (1) Look for the name Locally (parameter/local variable) in the function:

  E (2) Look for the name in an Enclosing function:

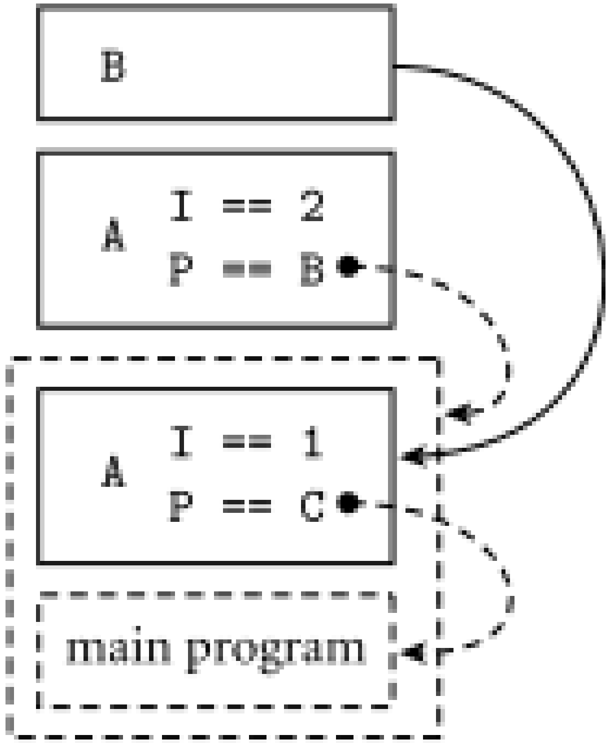
  G (3) Look for the name Globally (defined in the module outside the function):

  B (4) Look for the name in the Builtins module (imported automatically):

LEGB rule for bindings

- When we call bigger_than we execute its body, which creates a NEW function object bound to test; that function object can refer to v, which is a parameter bound to the argument passed to the call of bigger_than (in the Enclosing function).
- Then the return statement returns a reference to that new function object, which expect one argument (bound to its parameter x) when called.

```
def A(I, P):

    def B():
        print(I)

    # body of A:
    if I > 1:
        P()
    else:
        A(2, B)

def C():
    pass     # do nothing

A(1, C)      # main program
```



Dynamic Binding

LEGB rule for bindings

- Now, we can write the following

```
old     = bigger_than(60)
ancient = bigger_than(90)
print (old(10), old(70), old(90), ancient(70), ancient(95))
```

- Python prints

```
False True True False True
```

- Each assignment statement binds its name (old and ancient) to a different function object that is created and returned by calling the bigger_than function. Each function object remembers in v the argument used to call bigger_than. Finally, when we call each function, it uses the remembered value for v to compute its result: comparing each v against its argument bound to x.

LEGB rule for bindings

- In fact, we even could even have written something like

```
    print(bigger_than(60)(70))
```

- We know that bigger_than(60) calls bigger_than with the argument 60, which returns a result that is a reference to its inner function test; by writing (70) after that, we are again just calling a function: that inner function object, the one bigger_than(60) returned. When calling this function using the argument 70, it returns True.

LEGB rule for bindings

- Note that

```
 def f(x):
     return 2*x
```

is really just creating a name f and binding it to a new function object. So

```
     def test(x) :
         return x > v
```

binds the new function object to the local name test. And

```
     return test
```

returns the new function object currently bound to test (which remembers what value v has in the Enclosing scope, even after test is returned and bigger_than finishes executing).

LEGB rule for bindings

- Note the difference between the following, which both print 6.

```
    x = f(3)
    print(x)
```

and

```
    g = f
    print(g(3))
```

- A large part of this course deals with understanding functions better, including but not restricted to function calls: the main thing -but not the only thing-one does with functions; the other main things are passing functions as arguments to functions and returning functions from functions.

Functions vs Methods

LECTURE 9

Functions vs Methods

- Functions are typically called f(...) while methods are called on objects like o.m(...). Think of x ='candide' followed by calling `print(x.replace('d','p'))`
- In reality, a method call is just a special syntax to write a function call. The special "argument" o (normally arguments are written inside the parentheses) prefixes the method name. Functions and methods are related by what I call *"The Fundamental Equation of Object-Oriented Programming."*

```
    o.m(...)=type(o).m(o,...)
```

Functions vs Methods

On the right side

1) type(o) returns a reference to the class object o was constructed from.

2) .m means call the function m declared inside that class: look for

```
def m(self,...): .... in the class
```

3) pass o as the first argument to m: recall, that when defining methods in classes we write def m(self, ...); where does the argument matching the self parameter come from? It comes from the object o in calls like

```
o.m(...)
```

So, calling 'candide'.replace('d','p') is exactly the same as calling str.replace('candide','d','p'), because type('candide') returns a reference to the str class, which defines many methods, including the replace method.

Functions vs Methods

- We could equivalently write

```
from str import replace
```

and then call `replace('candide','d','p')`

- How well do you understand self (or your-self, for that matter:)? This equation is the key. I believe a deep understanding of this equation is the key to clarifying many aspects of object-oriented programming in Python (whose objects are constructed from classes). Just my two cents. But we will often return to this equation throughout this class. I've never seen any books that talk about this equation explicitly. We will revisit FEOOP when we spend a week discussing how to write sophisticated classes.

Functions vs Methods

- Oh, by the way, I must say that this equation is true, but not completely true.
- As we will later see: (a) if m is in the object's namespaces, it will be called directly (bypassing looking in o's class/type), and (b) when we learn about class inheritance, the inheritance hierarcy of a class provides other classes in which to look for m if it is not declared directly in o's class/type.
- But this equation is so simple and clear (once you understand it) and useful for tons of examples, it is worth memorizing, even if it is not completely accurate.

Lambda

LECTURE 10

Lambda

- Lambdas are used in expressions where we need a very simple function. A lambda represents a special function object. Instead of defining a full function (with a def), we can just use a lambda: after the word lambda comes its parameters separated by commas, then a colon followed by a single EXPRESSION that computes the value of a lambda (no "return" is needed, and the function cannot include control structures/statments, not even a sequence of statements).
- So, writing ...(lambda x,y : x+y)... in some context Is just like first defining

```
def f(x,y):
    return x+y
```

and then writing ...f... in the same context

Lambda

- A lambda produces an UNNAMED function object. For example, we can also write the following code, whose first line binds to the name f the lambda/function object, and whose second line calls the lambda via the name.

```
f = lambda x,y : x+y   # lambdas have one expression after
                       # without a return
print( f(1,3) )
```

and Python will print 4. I often put lambdas in parentheses, to more clearly denote the start and end of the lambda, for example writing

```
f =  (lambda x,y : x+y)
```

- Using this form, we can write code code above without defining f, writing just

```
print( (lambda x,y : x+y)(1,3) )
```

Lambda

- In my prompt module (MY PREFERRED WAY OF DOING PYTHON USER-INPUT), there is a function called for_int that has a parameter that specifies a boolean function (a function returning a boolean value: often called a predicate) that the int value must satisfy, to be returned by the prompt (otherwise the for_int function prompts the user again, for a legal value).
- That is, we pass a function object (without calling it) to prompt.for_int, which CALLS that function on the value the user enters to the prompt, to verify that the function returns True for that value.
- So, the following code fragment is guaranteed to store a value between 0 to 5 inclusive in the variable x. If the user enters a value like 7, an error will be printed and the user reprompted for the information.

Lambda

```
import prompt
x = prompt.for_int('Enter a value in [0,5]’,
                   is_legal=(lambda x : 0<=x<=5))
print(x)
```

- Execute this code in Python.
- In a later part of this lecture note, we will see how to use lambdas to simplify sorting lists (or simplify iterating through any data structures according to an ordering function).

Lambda

- Again, I often put lambdas in parentheses for clarity: see the prompt.for_int example above. But there are certain places where lambdas are required to be in parentheses: assume def g(a,b): ... and the lambda below will receive a 2-tuple as an argument and return the reversed 2-tuple. Calling g(1, (lambda x : x[1],x[0])) requires the lambda to be in parenthese, because without the parentheses it would read as g(1, lambda x : x[1],x[0]).
- In this function call, Python would think that x[0] was a third argument (not part of the lambda) to function g, which would raise an exception when called, because g defined only two parameters. Of course, we could also use different parentheses in the call ans write g(1, lambda x : (x[1],x[0])).

Lambda

- In a previous section (functions returning functions) we wrote the code

```
def bigger_than(v) :
    def test(x) :
        return x > v
    return test
```

- Because the test function is so simple, we can simplify this code by returning a lambda instead of defining/returning a named function:

```
def bigger_than(v) :
    return (lambda x : x > v)
```

- In each version of the bigger_than function, it returns a reference to a function object.

Parallel/Tuple/List Assignment

LECTURE 11

Parallel/Tuple/List Assignment

- Note that we can write code like the following: here both x,y and 1,2 are implicit tuples, and we can unpack them as follows

```
    x,y = 1,2
```

- In fact, you can replace the left-hand side of the equal sign by either (x,y) or [x,y] and replace the right-hand side of the equal sign by either (1,2) or [1,2] and x gets assigned 1 and y get assigned 2: even (x,y) = [1,2] works correctly.

Parallel/Tuple/List Assignment

- In most programming languages (including Java and C++), to exchange the values of two variables x and y we write three assignments (can you prove that writing just x = y followed by y = x fails to exchange these values?):

```
temp = x
x    = y
y    = temp
```

Parallel/Tuple/List Assignment

In Python we can write this code using one tuple assignment

```
    x,y = y,x
```

To do any parallel assignment, Python

- (a) computes all the expression/objects on the right (1 and 2 from the top)
- (b) binds the names on the left (x and y) to these value/objects (bind x to 2, then bind y to 1)

This is also called "sequence unpacking assignment". Note that x,y = 1,2,3 and x,y,z = 1,2 would both raise **`ValueError`** exceptions, because there are different numbers of names and values to assign to them. We will frequently use simple forms of parallel/unpacking assignment when looping through items in dictionaries (illustated below; used extensively later in this lecture note), but even more complicated forms are possible: for example.

Parallel/Tuple/List Assignment

```
l,m,(n,o) = (1, 2, [3,4])
print(l,m,n,o)
prints: 1 2 3 4
```

- Python also allows writing

```
a,*b,c = [1,2,3,4,5]
print(a,b,c)
```

which prints as: 1 [2, 3, 4] 5

- Here, * can preface one name; the name is bound to a list of any number of values, so as to correctly bind a and c.

Parallel/Tuple/List Assignment

- In fact, we can write complicated parallel assignments with multiple *s like

```
l,(*m,n),*o  = (1, ['a','b','c'], 2, 3,4)
print(l,m,n,o)
```

which prints as: 1 ['a', 'b'] c [2, 3, 4]

- Generally, sequence unpacking assignment is useful if we have a complex tuple/list structure and we want to bind names to its components, to refer to these components more easily by these names: each name binds to part of the complicated structure.

Parallel/Tuple/List Assignment

- As another example, if we define a function that returns a tuple

```
def reverse(a,b) :
    return (b,a)     # we could also write just return b,
```

we can also write `x,y = reverse(x,y)` to also exchange these values.

- Finally, we can use unpacking assignment in for loops: for example, we can write the following for loop to print the sum of each triple in the list

```
for i,j,k in [(1,2,3), (4,5,6), (7,8,9)]:
    print (i+j+k)
```

this is much simpler and clearer than using one name for the entire tuple

```
for t in [(1,2,3), (4,5,6), (7,8,9)]:
    print (t[0] + t[1] + t[2])
```

Parallel/Tuple/List Assignment

- The loop above assigns i, j, and k the three values in each 3-tuple in the list that the for loop is iterating over. We will see more about such assignments when iterating though items in dictionaries. A preview is

```
 for k,v in d.items():    # d is any dictionary
     print(k,'->',v)      # print its key and
                          # value pairs
                          #  (abbreviated k,v)
```

which prints each key and its associated value for each item (each key/value association) that we are iterating through in a dictionary.

Parallel/Tuple/List Assignment

```
d = {'a':1, 'b':2, 'c':3, 'd':4}
for k,v in d.items():
    print(k,'->',v)
```

prints

```
d -> 4
a -> 1
c -> 3
b -> 2
```

although the keys/values may print in any order

Parallel/Tuple/List Assignment

- This is simpler than the following loop, which iterates over the tuples produced when iterating over d.items().

```
for t in d.items():             # d is any dictionary
    print(t[0],'->',t[1])   # print its key (t[0])and
                            # value (t[1]) pairs
```