CSC 148
Lecture 3

Dynamic Typing, Scoping, and Namespaces

Recursion

Announcements

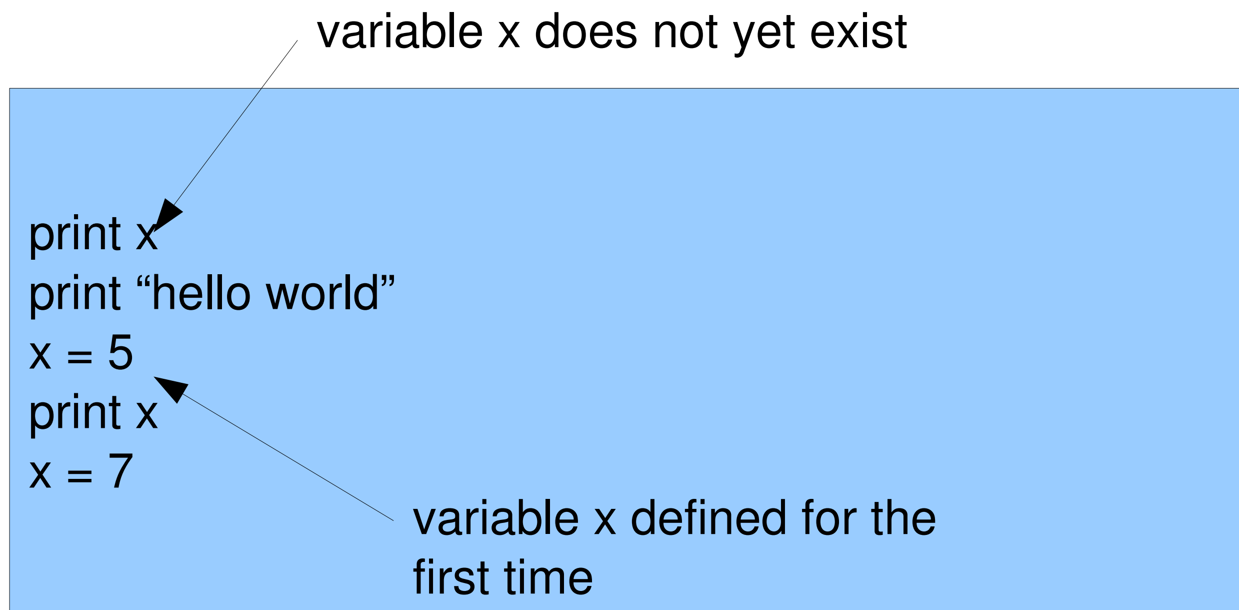
- Python Ramp Up Session
  - Monday June 1st, 1-5pm. BA3195
  - This will be a more detailed introduction to the Python language than we have time to do in class.
  - Attendance is optional, and recommended if you are struggling with the Python language.
- Assignment 2 will be posted Friday afternoon.

Names vs. Values

- A name (e.g. variable) is just a way of identifying some object (value) that you want to keep track of
- For example, I may assign the value “Indiana Jones” or “Star Trek” to the variable named `movie`.
  - `movie` is the name of a variable
  - “Indiana Jones” and “Star Trek” are the string objects that can be assigned to the variable.

“Defining” Variables in Python

- Variables are never explicitly defined like in other languages
- A variable is “created” when you first assign a value to it



Dynamic Typing

- Python is a dynamically typed language
  - “type checking” is done at runtime
- You don't have to declare the type of a variable statically when you write your code.
  - This is different from languages like Java, C, C++.

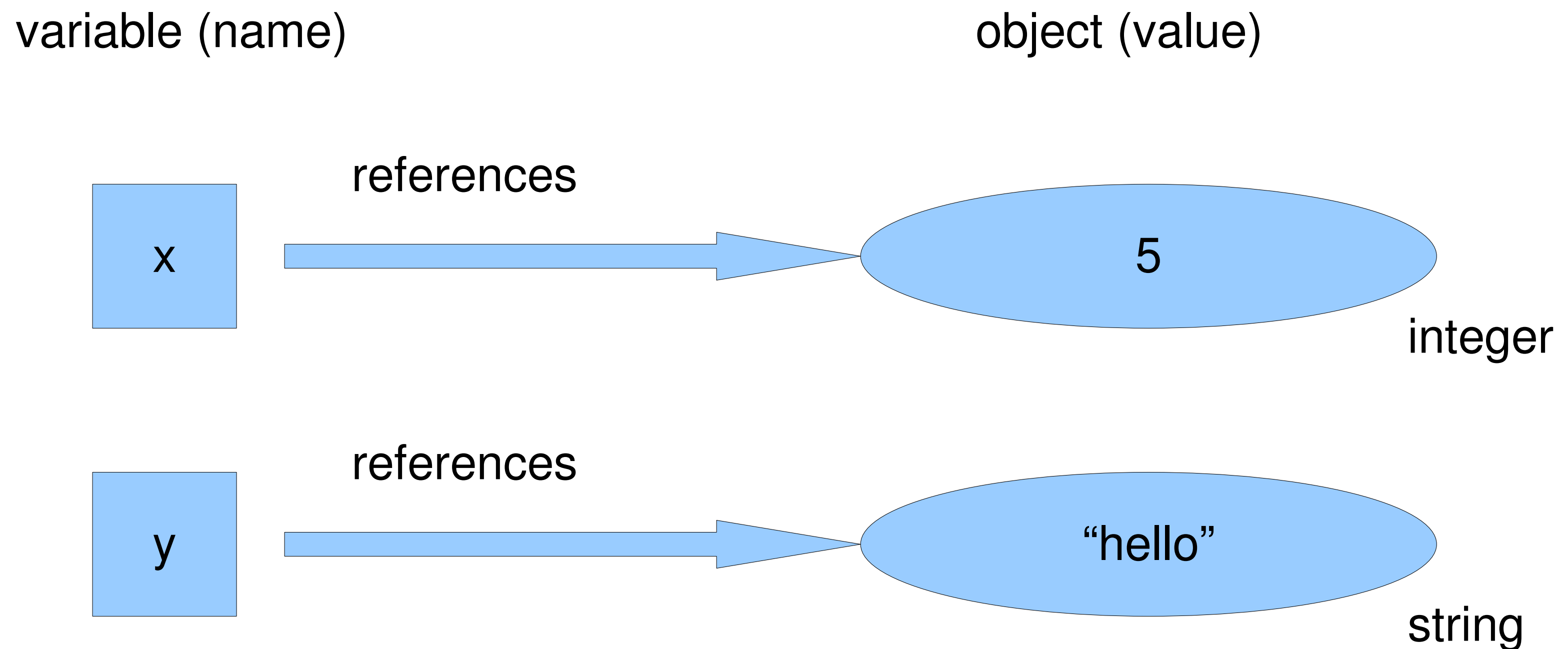
Python Variables and Dynamic Typing

- Variables have no type information associated with them
- But... if this is true, how does the following “TypeError” happen?

```
>>> x= 5
>>> y = "hello"
>>> x + y
Traceback (most recent call last):
  File "<string>", line 1, in <string>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Python Variables and Dynamic Typing

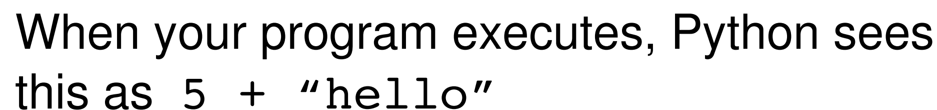
- Type information is associated with the object stored in the variable.



Python Variables and Dynamic Typing

- Whenever a variable appears in your code, it is "replaced" by the object (value) that it is referencing.

```
>>> x= 5
>>> y = "hello"
>>> x + y
```



What's in a name?

- A variable is essentially a “name” (label, identifier).
- You assign values to a variable (name) by using assignment statements
  - This is technically known as **name binding**. You are binding an object value to a name
- Variables are not the only entities in Python that have names. There's also:
  - classes, methods, modules, functions

Functions

- The “def” statement is an executable statement.
- When it's executed, the name of the function is bound to the function definition (i.e., the function object)

```
def func():
    # do stuff here
```

Modules and Classes

- “import” statements bind a module object to a module name
- “class” statements bind a class object to a class name
- Terminology alert: do not to confuse “class object” with an object that is an instance of some class.

Hang on a second...

- If a class definition (or function definition, import statement, etc) is an executable statement that binds a name to an object, doesn't this mean that I can put it anywhere in my code?
- Yes. Well... anywhere within reason. Your program still has to be valid python.
- But this does mean you can put it in places you might have not expected. (... especially if you're used to a language like C or Java)

Hang on a second ...

- And can I treat class names (function names, module names) just like variable names?
- Yes. These names (if bound) have objects that they reference. You can access these objects just like you can an object assigned to a variable name.

Example

- Lets see an example in Wing!

Where else can names be bound?

- Function parameters in a function header
- for loop headers
- except clause headers

Namespaces

- How are names and their bindings kept track of?
- Namespace: a mapping from names to objects
  - think of it as a dictionary
  - there can be different namespaces: e.g. the variable x can be bound to different objects in different namespaces.

Scope

- How does Python distinguish between different namespaces?
- **Scope:** Technically, a “region” of the program that has a distinct namespace.
- Sometimes we'll talk about the “scope of a name”: A region of the program in which a particular (name, object) binding “lives”.
- Every namespace belongs to a scope, and every scope has a namespace.
  - “Namespace” and “Scope” are sometimes used interchangeably.

Scopes in Python

- The scopes in python are as follows:
  - the local scope (e.g. in a function call)
  - enclosing scope (e.g., in an enclosing function)
  - global scope (module scope)
  - built-in scope

The Global Scope and Modules

- The namespace of the enclosing module resides in the global scope.
- Terminology alert: “Global” doesn't mean that it's global to **everything**. Global means global to a module.
- The namespace of other modules can be accessed by using the import statement. Any name in the other module's namespace is an attribute of the module object.

Name binding and Namespaces

- When a name is bound, what namespace will the binding be stored in?
- Generally, it is stored in the namespace associated with the scope where the binding takes place
  - e.g. in an assignment statement in a function call, the binding is associated with the namespace for the local scope of the function
  - if the assignment statement is at the module level, the binding is associated with namespace for the global module scope

Name Resolution

- Name resolution: figuring out which namespace to use to look up a reference to a name
- LEGB rule: When a name is referenced, python looks it up in the following order:
  - the Local (function) scope
  - the Enclosing function scope
  - the Global (module) scope
  - the Built-in scope

Examples

- Lets see some examples in Wing!

Recursion

- The problem is too big/too complicated! I don't know how to solve it!
- If I could solve a slightly smaller problem, I would be able to use that solution to come up with the solution to the original problem!
- But I don't know how to solve that slightly smaller problem!
- If I could solve a problem that's even a bit smaller than that one ...

Recursion

- Recursion: a method for solving problems that involves
  - breaking a problem down into smaller and smaller subproblems until you get a small enough problem that can be solved trivially
  - using the solution to the smaller problem to solve the larger problem

Recursion – The Base Case

- The “small enough problem that can be solved trivially” is known as the **base case.**
- Recursion ends when the base case has been reached.

Thinking Recursively

- The idea behind recursion is very easy to state
- Thinking recursively takes some practice
- Lets spend some time working on a few examples

Determining the sum of a list of integers

- Wing!

Factorial Function

- The expression "n!" is read as "n factorial". It is recursively defined for non-negative n as follows:
  - n! = n * (n-1)!   for n >= 1
  - 0! = 1   (base case)
- Lets come up with both iterative and recursive solutions in Wing!

A Well-known Sequence

- 0,1,1,2,3,5,8,13,21,34, ...
- What's the pattern?
- How do we define it recursively?

Fibonacci Sequence

- $F(n) = 0$ if $n = 0$
- $F(n) = 1$ if $n = 1$
- $F(n) = F(n-1) + F(n-2)$ if $n > 1$
- Implementing a recursive function naively can sometimes be inefficient!
- Lets compare iterative and (naïve) recursive implementations in Wing
- Iterative version **much** faster than recursive. Why?

Towers of Hanoi

- Three pegs
- A number of different sized discs, all stacked from largest disc (at bottom) to smallest (at top) on peg 1.
- Move the discs from peg 1 to peg 3, using peg 2, without ever putting a larger disc on top of a smaller disc

Towers of Hanoi

- How do we modify this function we wrote in Wing (same as the function in your text) so that it returns the total number of moves required?