Python

CSE 216 : Programming Abstractions

Department of Computer Science Stony Brook University

Dr. Ritwik Banerjee

Python scripts and modules

- The interactive shell is good for a quick check involving a few lines of code.
- For your code to last longer than one session, you will be writing your code typically in a `.py` file. This is commonly called a python **script**.
- Longer programs are split across multiple files.
  - Easier maintenance
  - Code reusability
- The 'proper' way to do this is to put definitions in file(s), and use them in a script (or the shell). Such a file is called a **module**.
  - Definitions from a module can be `import`ed into other modules.

Modules

- A module is a .py file containing definitions and functions.
- The module's name is the file name, minus the .py extension.
  - Within a module, this name is available as the built-in global variable __name__.

```python
def fib(n): # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

fibonacci.py

```
>>> import fibonacci
>>> fibonacci.fib(120)
0 1 1 2 3 5 8 13 21 34 55 89
>>> fibonacci.__name__
'fibonacci‘
>>> # assign a local name
>>> fib = fibonacci.fib
>>> fib(120)
0 1 1 2 3 5 8 13 21 34 55 89
```

The ‘import’ statement

1. `import fibonacci`
    - Use `<modulename>.<itemname>` in subsequent code.
2. `from fibonacci import fib, fib2`
    - Use `itemname` directly in subsequent code.
    - But `modulename` is not defined in this code.
3. `from fibonacci import *`
    - Import all names except those beginning with an underscore.
    - ! It introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.
    - ! Code is less readable (due to the above reason).
4. `import fibonacci as fibo`
    - Directly bind an easier name to the module.

Running a module as a script

```python
if __name__ == "__main__":
    import sys
    fibonacci(int(sys.argv[1]))
```

- By adding this code at the end of our module, we can use it as a script as well as an importable module.
  - Because the code that parses the command line only runs if the module is executed as the "main" file.
  - Simply importing the module will not execute the code.
  - We need to run the module as `python fibonacci.py <arguments>`

Object-oriented programming in Python

Names & Objects

- Recall **aliasing**: multiple names (in multiple scopes … we are getting there soon) can be *bound* to the same object.
- Aliasing can usually be ignored when dealing with immutable objects like numbers, strings, tuples, etc.
- For mutable objects (*e.g.*, lists and dictionaries), however, aliasing offers surprising benefits.
- Aliasing behaves ‘somewhat’ like pointers:
  - it is ‘cheap’ to pass an object, since only a pointer is passed by the implementation.
  - if a function modifies an object passed as an argument, the modification can be seen by the caller.

Namespaces

- A **namespace** is a mapping from names to objects.
- Behind the scenes, namespaces are implemented as dictionaries
  - this is usually not noticeable to the programmer, and
  - this implementation detail may change in future versions of Python.
- Examples of namespaces:
  - built-in function and exception names (*e.g.*, `abs()`, `KeyError`)
  - global names in a module
  - local names in a function

! There is no relation between names in different namespaces.
  - Two different modules may both define a function with the same name.
  - The programmer must use the module's name as a prefix to use the correct function.

Namespaces

- Namespaces are created at different times (recall *binding* of names to objects):
  - the namespace of a built-in name is created when the Python interpreter starts; this is never deleted (these names live in a module called `builtins`)
  - the global namespace for a module is created when the module definition is read
  - unless specifically deleted, a module namespace exists until the interpreter quits.
- The statements executed at the top-level are considered part of a module called **`__main__`**.
  - `__main__` is the `__name__` of the top-level module
  - since this module is read as soon as the execution starts, the statements executed in this module have their own global namespace throughout

```python
if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Attributes

- In Python, an **attribute** usually indicates any name that follows a ‘dot’
  - in `p.name`, the field `name` is an attribute of the object p
  - in `fibonacci.fib`, the function `fib` is an attribute of the module `fibonacci`
- A module’s attributes and the set of global names defined in the module share the exact same namespace.
  - In this sense, the complete set of attributes of an object ‘define’ a namespace.
- An attribute can be read-only, or writeable.
- Unlike Java, Python allows you to delete an attribute by using the `del` statement:
  - The following removes the attribute from the namespace:

```
del <module_name>.<attribute_name>
```

Scopes

Qualified and unqualified references

- If there is a function `func()` in a module `m` inside a package `x.y.z`, then `x.y.z.m.func` is its **fully qualified** reference/name, while just `func` is its **unqualified** name/reference.
- A **scope** is a *textual region* of a program where a namespace is accessible through unqualified reference.
- The scope is determined statically, but used dynamically.
- During execution, there are multiple nested scopes whose namespaces are directly accessible (in order of search):
  - the innermost scope containing local names
  - the scopes of enclosing functions
  - the scope containing the current module's global names
  - the outermost scope, which is the namespace containing built-in names

Scopes

- The `nonlocal` statement can be used to re-bind a variable found outside the innermost scope.
  - ! If not declared nonlocal, these variables are read-only.
  - ! If you try to write such a variable, you will end up creating a new local variable in the innermost scope while the outer variable with the same name remains unchanged.

1. **Innermost scope** containing **local names**
2. Scopes of **enclosing functions**
3. Scope containing the **current module's global names**
4. **Outermost scope**, which is the namespace containing **built-in names**

Scopes

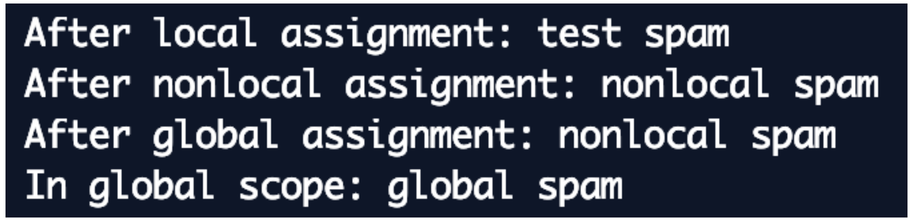
- If a name is declared `global`, then all references and assignments go directly to the middle scope containing the module's global names.
  - ❗ If the `global` statement is not used, names are always assigned to the innermost scope.
  - ❗ Any operation that introduces a new name, *always* uses the local scope.
  - ❗ Remember that an assignment statement doesn't *copy* any data. It simply binds a name to an object.

1. **Innermost scope** containing **local names**
2. Scopes of **enclosing functions**
3. Scope containing the **current module's global names**
4. **Outermost scope**, which is the namespace containing **built-in names**

Scopes
(*in less abstract terms*)

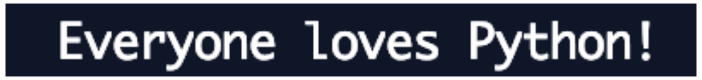
- In many languages, variables are treated as global if not declared otherwise.
- Python has the opposite approach. Why?
  - Because global variables should be used only when absolutely necessary.
  - In most cases, if you are tempted to use a global variable, it is better to (i) have a formal parameter to get a value into a function, or (ii) return a value to get it out of a function.
  - Variables inside a function are local to this function by default.
  - The function body is the scope of such a variable.
- Remember: variables can't be declared in the way they are declared in programming languages like Java.
  - In Python, they are implicitly declared by defining them, *i.e.*, the first time you assign a value to a variable.
  - This variable is declared and has automatically the data type of the object which has to be assigned to it.

```python
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```



```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

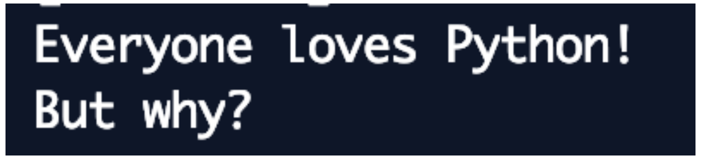
```python
def f():
    print(s)

s = "Everyone loves Python!"
f()
```



```
Everyone loves Python!
```

- Before calling the function f(), there is no local variable s (*i.e.*, no object is bound to the name).
- So, the value of the global variable is used.
- What will happen if we change the value of the variable inside the function?

```python
def f():
    s = "Everyone loves Python!"
    print(s)

s = "But why?"
f()
print(s)
```



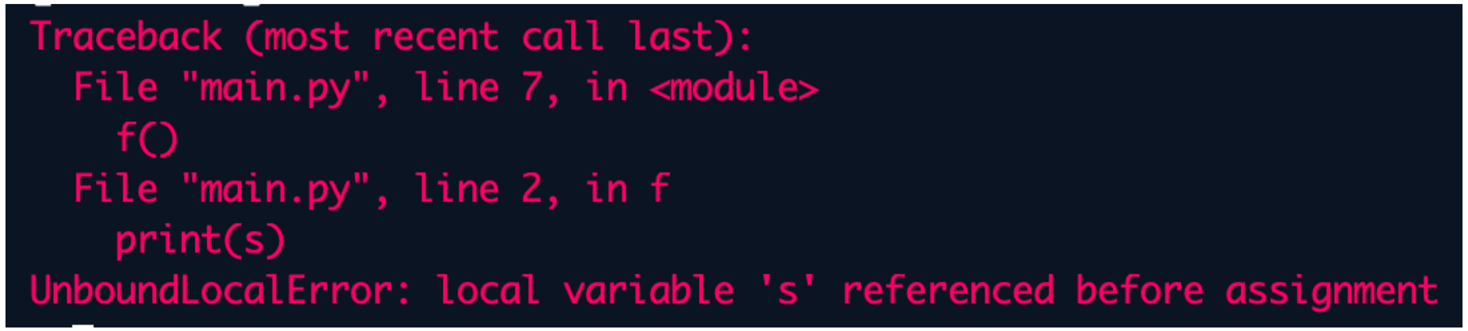
```
Everyone loves Python!
But why?
```

- Before calling the function f(), there is no local variable s (*i.e.*, no object is bound to the name).
- So, the value of the global variable is used.

Mixing global and local scope?

```python
def f():
    print(s)
    s = "Everyone loves Python!"
    print(s)

s = "But why?"
f()
print(s)
```

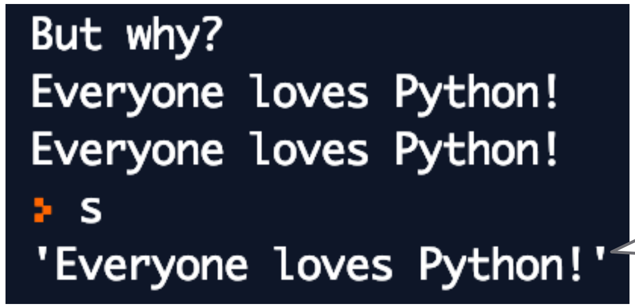
Print the global value, then reassign locally to get the local value?



```
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    f()
  File "main.py", line 2, in f
    print(s)
UnboundLocalError: local variable 's' referenced before assignment
```

global

```
def f():
    global s
    print(s)
    s = "Everyone loves Python!"
    print(s)

s = "But why?"
f()
print(s)
```



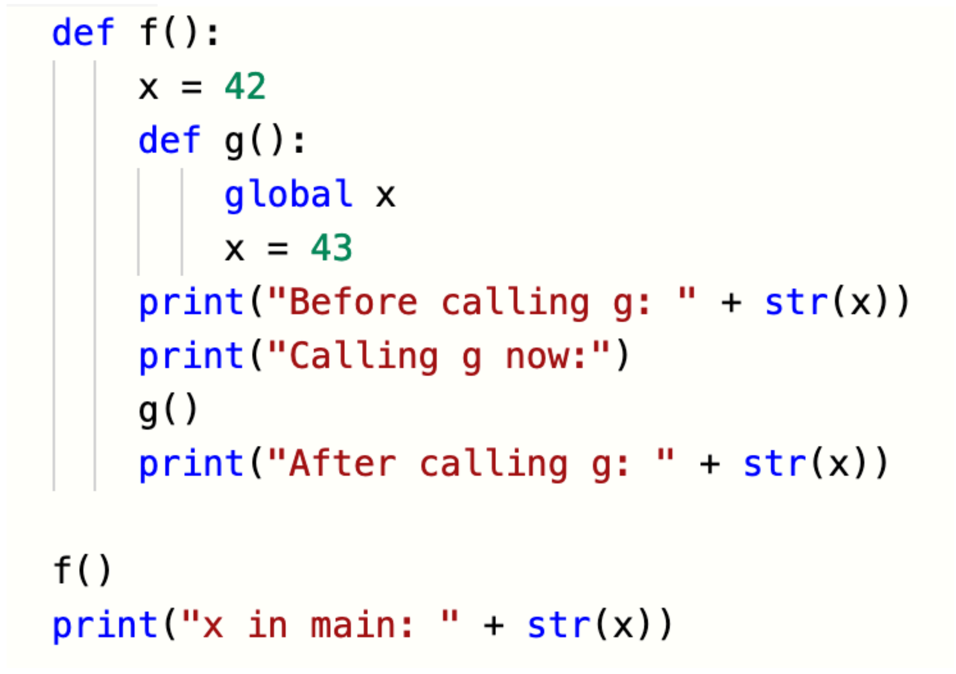
```
But why?
Everyone loves Python!
Everyone loves Python!
> s
'Everyone loves Python!'
```

The original value has been overwritten.

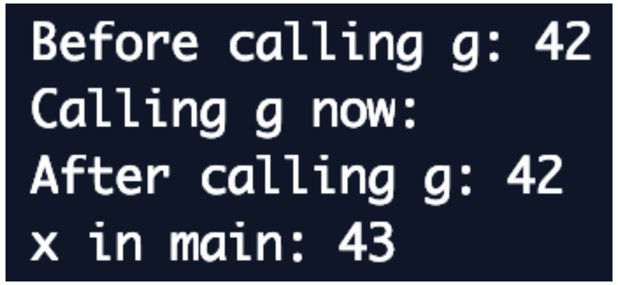
- A variable can't be both global and local inside a function.
  - Which is why the previous code crashed!
- We can, however, explicitly specify that even inside the function, we want to use the **`global`** variable.

global

- The global statement inside of the nested function does not affect the variable x of the function f (value is 42).
- After calling f() a variable x exists in the module namespace and has the value 43.
- Conclusion (again): a variable defined inside of a function is local unless it is explicitly marked as global.



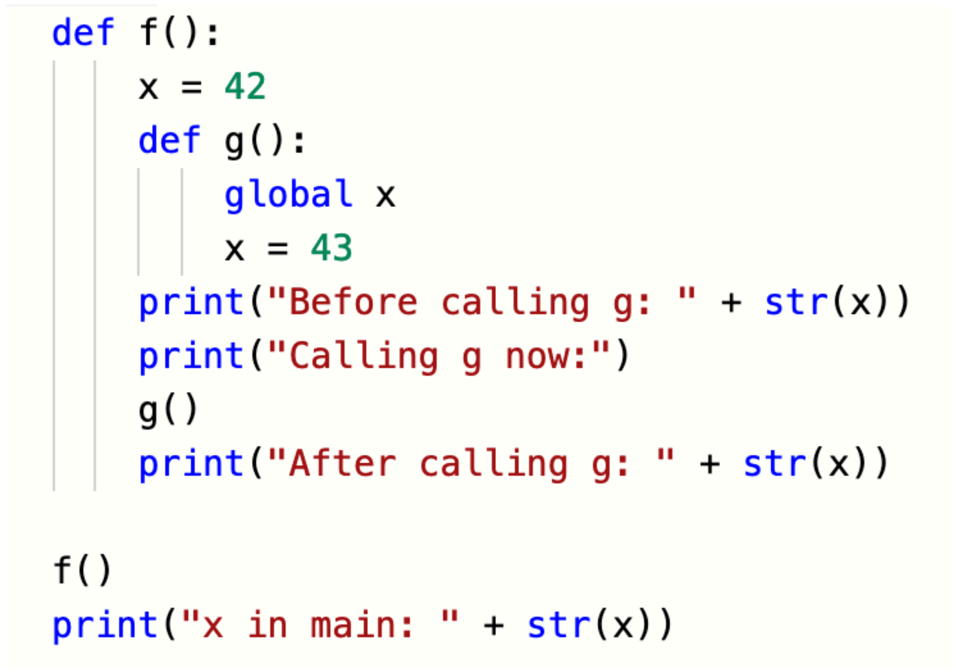
```python
def f():
    x = 42
    def g():
        global x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))

f()
print("x in main: " + str(x))
```



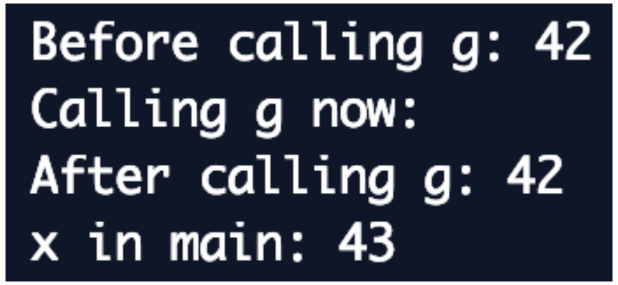
```
Before calling g: 42
Calling g now:
After calling g: 42
x in main: 43
```

nonlocal

- We can refer a variable name in any enclosing scope, but we can only rebind variable names (i) in the local scope by assigning to it, or (ii) in the module-global scope by using a global declaration.
- We need a way to access variables of other scopes as well.
- This is done by the **nonlocal** statement.



```python
def f():
    x = 42
    def g():
        global x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))

f()
print("x in main: " + str(x))
```



```
Before calling g: 42
Calling g now:
After calling g: 42
x in main: 43
```

First-class everything

- Python was designed according to the principle “first-class everything”.

“One of my goals for Python was to make it so that all objects were ‘first class.’ By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, and so on) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth.”

~ Guido van Rossum

Classes

- Classes provide a means of bundling data and functionality together.
- Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made.
- Each class instance can have attributes attached to it for maintaining its state.
- Class instances can also have methods for modifying its state.
- Compared with most other programming languages, Python’s class mechanism adds classes with a minimum of new syntax and semantics.

A minimal class in Python

```
class Robot:
    pass
```

- A class has two parts: a **header**, and a **body**.
- We can already start using this class:

```
class Robot:
    pass

if __name__ == "__main__":
    x = Robot()
    y = Robot()
    y2 = y
    print(y == y2) # True
    print(y == x)  # False
```

Encapsulation

- That is, there is no restriction on accessing a variable or calling a method in a Python program.
- All member variables and methods are public by default in Python!
- So, to make a member public, you do no extra work:

```
class Cup:
    def __init__(self):
        self.color = None
        self.content = None

    def fill(self, beverage):
        self.content = beverage

    def empty(self):
        self.content = None
```

Encapsulation

- Having a member protected or private is a matter of convention in Python.
  - Protected members are prefixed with a single underscore (e.g., _content)

```
class Cup:
    def __init__(self):
        self.color = None
        self._content = None

    def fill(self, beverage):
        self._content = beverage

    def empty(self):
        self._content = None
```

- You can still access the variable from outside the class and assign a value: `cup._content = "tea"`

Encapsulation

- Having a member protected or private is a matter of convention in Python.
  - Private members are prefixed with two underscores (e.g., __content)
  - They can still be accessed from outside; the double underscore simply means that they *should not be* accessed or modified from outside.

```
class Cup:
    def __init__(self):
        self._color = None.    # protected
        self.__content = None # private

    def fill(self, beverage):
        self.__content = beverage

    def empty(self):
        self.__content = None
```