Special Topics in Computer Science- CSC 4992

Managing the Namespace

What Is the Namespace?

- The *namespace* of a program has to do with the arrangement of names in a program
- These names refer to modules, data values, functions, methods, and data types
- Best to minimize the number of names in a program, but still have to organize!

Example Names

- Modules – **`math`**, **`random`**, **`doctor`**, **`generator`**
- Data types – **`int`**, **`float`**, **`str`**, **`list`**
- Variables – **`average`**, **`qualifiers`**, **`replacements`**
- Functions – **`print`**, **`input`**, **`sum`**, **`math.sqrt`**, **`reply`**
- Methods – **`append`**, **`split`**, **`lower`**

Properties of a Name

- *Binding time* - the point at which a name is defined to have a value
- *Scope* - the area of program text in which a name has a particular value
- *Lifetime* - the span of time during which a name has a particular value

The Scope of a Name

- A name's *scope* is the area of program text within which it has a particular value
- This value will be visible to some parts of a program but not to others
- Allows a single name to have different meanings in different contexts

The Scope of a Module Variable

```
qualifiers = ['Why do you say that ',
              'You seem to think that ',
              'Did I just hear you say that ']

for q in qualifiers: print(q)

def reply(sentence):
    return random.choice(qualifiers) + changePerson(sentence)
```

A *module variable* is visible throughout the module, even within a function definition

Module variables may be reset with assignment, but not within a function definition

The Scope of a Module Variable

```
qualifiers = ['Why do you say that ',
              'You seem to think that ',
              'Did I just hear you say that ']

def reply (sentence):
    qualifiers=sentence
    return(qualifiers)

print(reply("Hi there"))
print(qualifiers)
```

```
Hi there
['Why do you say that ', 'You seem to think that ', 'Did I just hear you
say that ']
```

The Scope of a Module Variable

```
qualifiers = ['Why do you say that ',
              'You seem to think that ',
              'Did I just hear you say that ']

def reply (sentence):
    qualifiers.append(sentence)
    return(qualifiers)

print(reply("Hi there"))
print(qualifiers)
```

```
['Why do you say that ', 'You seem to think that', 'Did I just hear you say that ','Hi there']
['Why do you say that ', 'You seem to think that', 'Did I just hear you say that ','Hi there']
```

The Scope of a Parameter

```
def reply(sentence):
    return random.choice(qualifiers) + changePerson(sentence)

def changePerson(sentence):
    oldlist = sentence.split()
    newlist = []
    for word in oldlist:
        newlist.append(replacements.get(word, word))
    return " ".join(newlist)
```

A *parameter* is visible throughout its function definition but not outside of it

The Scope of a Temporary Variable

```
def mySum(lyst):
    total = 0
    for number in lyst:
        total += number
    return total

def average(lyst):
    total = mySum(lyst)
    return total / len(lyst)
```

A *temporary variable* is visible within its function definition but not outside of it

Temporaries may be reset with assignment

The Scope of a Temporary Variable

```
def changePerson(sentence):
    oldlist = sentence.split()
    newlist = []
    for word in oldlist:
        newlist.append(replacements.get(word, word))
    return " ".join(newlist)
```

A *temporary variable* is visible throughout its function definition but not outside of it

Temporaries may be reset with assignment

The Scope of a Loop Variable

```
def changePerson(sentence):
    oldlist = sentence.split()
    newlist = []
    for word in oldlist:
        newlist.append(replacements.get(word, word))
    return " ".join(newlist)
```

A *loop variable* is like a temporary variable but is visible only in the body of the loop

Don't *ever* reset loop variables with assignment!

How Scope Works

```
x = "module"

def f():
    x = "temporary"
    print(x)

print(x)      # Outputs module
f()           # Outputs temporary
print(x)      # Outputs module
```

A temporary variable and a module variable can have the same name, but they refer to different storage spaces containing possibly different values

Name Conflicts

```
x = "module"

def f():
    x += "temporary" # Error: reference before assignment
    print(x)

print(x)     # Outputs module
f()          # Should output moduletemporary, but generates error
print(x)     # Outputs module
```

Python thinks you're referencing a temporary variable (the second **x**) before you define it

Limits on Module Scope

```
history = []

for s in history: print(s)

history += ["All computer scientists are cool!"]

def reply(sentence):
    answer = random.choice(qualifiers) + changePerson(sentence)
    history = history + [sentence] #Error
    return answer
```

```
# UnboundLocalError: local variable 'history' referenced
before assignment
```

Python thinks that **`history`** is a temporary variable within the **`reply`** function

Limits on Module Scope

```
history = []

for s in history: print(s)

def reply(sentence):
    answer = random.choice(qualifiers) + changePerson(sentence)
    history.append(sentence)  # This is a better practice
    return answer
```

Better to reference the variable's value and mutate this value than to reset the variable itself

Question

```
def f():
   s = "I am globally not known"
   print (s)


f()
print (s)
```

NameError: name 's' is not defined

Question

```
def f():
  print (s)
  s = "Me too."
  print (s)


s = "I hate spam."
f()
print (s)
```

UnboundLocalError: local variable 's' referenced before assignment

Global variable

Variables are local, if not otherwise declared.

```
def f():
  global s
  print (s)
  s = "That's clear."
  print (s)

s = "Python is great!”
f()
print (s)
```

Python is great!
That's clear.
That's clear.

Question

```
def foo(x, y):
  global a
  a = 42
  x,y = y,x
  b = 17
  c = 100
  print (a,b,x,y)

a,b,x,y = 1,15,3,4
foo(17,4)
print (a,b,x,y)
```

42 17 4 17

42 15 3 4

Question

```
def atm_to_mbar(pressure):
    return pressure * 2
def mbar_to_mmHg(pressure):
   return pressure * 4


in_atm = 2
in_mbar = atm_to_mbar(in_atm)
in_mmHg = mbar_to_mmHg(in_mbar)
print("in_mbar: ",in_mbar)
print("in_mmHg: ",in_mmHg)
```

in_mbar: 4
in_mmHg: 16

Question

```
def func1(a):
    a[0] = "bbb"
    a[1] = a[1] + 1


args = ["aaa",10]
func1(args)
print (args[0],args[1])
```

List is mutable!

bbb 11

Binding Time of Module Variables

```
qualifiers = ['Why do you say that ',
              'You seem to think that ',
              'Did I just hear you say that ']

for q in qualifiers: print(q)

def reply(sentence):
    return random.choice(qualifiers) + changePerson(sentence)
```

A module variable is defined to have a value when the module is loaded into the Python interpreter

Its *binding time* is thus at load time

Binding Times of Temps and Params

```
def changePerson(sentence):
    oldlist = sentence.split()
    newlist = []
    for word in oldlist:
        newlist.append(replacements.get(word, word))
    return " ".join(newlist)

changePerson(input("Enter a sentence:"))
```

The *binding time* of a temporary variable is within a particular call of the function

A parameter is bound to a value after the argument is evaluated before a particular call of the function

The Lifetime of a Variable

- The *lifetime* of a module variable is the span of time during which the module is active

- The *lifetime* of a parameter or a temporary variable is the span of time during which a function call is active

Default (Keyword) Arguments

- Arguments provide the function's caller with the means of transmitting information to the function
- Programmer can specify optional arguments

```
def <function name>(<required args>,
                    <key-1> = <val-1>, … <key-n> = <val-n>)
```

  definition:
  - Following the required arguments are one or more **default or keyword arguments**
  - When function is called with these arguments, default values are overridden by caller's values

Default (Keyword) Arguments (continued)

```
def repToInt(repString, base):
    """Converts the repString to an int in the base
    and returns this int."""
    decimal = 0
    exponent = len(repString) - 1
    for digit in repString:
        decimal = decimal + int(digit) * base ** exponent
        exponent -= 1
    return decimal
```

```
def repToInt(repString, base = 2):

>>> repToInt("10", 10)
10
>>> repToInt("10", 8)    # Override the default to here
8
>>> repToInt("10", 2)    # Same as the default, not necessary
2
>>> repToInt("10")       # Base 2 by default
2
>>>
```

Default (Keyword) Arguments (continued)

- The default arguments that follow can be supplied in two ways:
  - **By position**
  - **By keyword**

```
def example(required, option1 = 2, option2 = 3):
    print required, option1, option2

>>> example(1)                  # Use all the defaults
1 2 3
>>> example(1, 10)              # Override the first default
1 10 3
>>> example(1, 10, 20)          # Override all the defaults
1 10 20
>>> example(1, option2 = 20)    # Override the second default
1 2 20
>>> example(1, option2 = 20, option1 = 10)  # Note the order
1 10 20
```

Higher-Order Functions (Advanced Topic)

- A **higher-order function** expects a function and a set of data values as arguments
  - Argument function is applied to each data value and a set of results or a single data value is returned
- A higher-order function separates task of transforming each data value from logic of accumulating the results

Functions as First-Class Data Objects

- Functions can be assigned to variables, passed as arguments, returned as the values

```
>>> abs                             # See what a function looks like
<built-in function abs>
>>> import math
>>> math.sqrt
<built-in function sqrt>
>>> f = abs                         # f is an alias for abs
>>> f                               # Evaluate f
<built-in function abs>
>>> f(-4)                           # Apply f to an argument
4
>>> funcs = [abs, math.sqrt]        # Put the functions in a list
>>> funcs
[<built-in function abs>, <built-in function sqrt>]
>>> funcs[1](2)                     # Apply math.sqrt to 2
1.4142135623730951
```

Functions as First-Class Data Objects (continued)

- Passing a function as an argument is no different from passing any other datum:

```
>>> def example(functionArg, dataArg):
        return functionArg(dataArg)

>>> example(abs, -4)
4
>>> example(math.sqrt, 2)
1.4142135623730951
>>>
```

```
example(max,(3,4)
```

```
4
```

- Apply a function to its arguments by passing it and a sequence of its arguments to the **`example`** function:

Mapping

- **Mapping** applies a function to each value in a list and returns a new list of the results

```
>>> words = ["231", "20", "-45", "99"]
>>> map(int, words)           # Convert all strings to ints
[231, 20, -45, 99]
>>> words                     # Original list is not changed
['231', '20', '-45', '99']
>>> words = map(int, words)   # Reset variable to change it
>>> words
[231, 20, -45, 99]
>>>
```

Map Problem

Goal: given a list of three dimensional points in the form of tuples, create a new list consisting of the distances of each point from the origin

Loop Method:

- distance(x, y, z) = sqrt(x**2 + y**2 + z**2)
- loop through the list and add results to a new list

Map Problem

```
from math import sqrt

points = [(2, 1, 3), (5, 7, -3), (2, 4, 0), (9, 6, 8)]

def distance(point) :
    x, y, z = point
    return sqrt(x**2 + y**2 + z**2)

distances = list(map(distance, points))
```

Filtering

- When **filtering**, a function called a **predicate** is applied to each value in a list
  - If predicate returns **`True`**, value is added to a new list; otherwise, value is dropped from consideration

```
>>> def odd(n): return n % 2 == 1

>>> filter(odd, range(10))
[1, 3, 5, 7, 9]
>>>
```

Reducing

- When **reducing**, we take a list of values and repeatedly apply a function to accumulate a single data value

```
>>> def add(x, y): return x + y

>>> def multiply(x, y): return x * y

>>> data = [1, 2, 3, 4]
>>> reduce(add, data)
10
>>> reduce(multiply, data)
24
>>>
```

Using `lambda` to Create Anonymous Functions

- A **`lambda`** is an **anonymous function**
  - When the **`lambda`** is applied to its arguments,

```
lambda <argname-1, ..., argname-n>: <expression>
```

```
>>> data = [1, 2, 3, 4]
>>> reduce(lambda x, y: x + y, data)    # Produce the sum
10
>>> reduce(lambda x, y: x * y, data)    # Produce the product
24
```

```
def sum(lower, upper):
    """Returns the sum of the numbers from lower to upper."""
    if lower > upper:
        return 0
    else:
        return reduce(lambda x, y: x + y,
                      range(lower, upper + 1))
```

Map Example

```
nums = [0, 4, 7, 2, 1, 0 , 9 , 3, 5, 6, 8, 0, 3]

nums = list(map(lambda x : x % 5, nums))

print(nums)
#[0, 4, 2, 2, 1, 0, 4, 3, 0, 1, 3, 0, 3]
```

Filter Example

```
nums = [0, 4, 7, 2, 1, 0 , 9 , 3, 5, 6, 8, 0, 3]

nums = list(filter(lambda x : x != 0, nums))

print(nums)       #[4, 7, 2, 1, 9, 3, 5, 6, 8, 3]
```