Intro to Computer Science

Previous

- Functions

Readings	
Gaddis	• Chapter 5

Next

- Functions (continued)

Readings	
Gaddis	• Chapter 5

Global versus local

Global variables

```
x = 10
def my Fun():
   print(x)
print(x)
```

Both uses are okay because `x` has *global* scope

Local variables

```
def myFun():
   x = 10
   print(x)
print(x)
```

`x` now has *local* scope:

- inside myFun it's defined;
- outside myFun it's not

Parameter (names) are local!

- Parameter assignment is like variable assignment in a new program
  - Acceptable to reuse existing names
  - Think of parameters as a new instance

```
x = 10
def myFun(x):
    print(x)

myFun(x)
```

- Having used the name x before is okay!
- These x's are different

```
x = 10
def myFun(x):
    print(x + ' World')

myFun('Hello')
```

- The parameter can share a name and have a different type!

Return values

- Use the keyword `return` to return (a value) from a function

Returns nothing	Returns an integer	Returns a list
`def voidFun():` `  return`	`def intFun():` `  return 10`	`def listFun():` `  return [1,2,3]`

Equivalent to not having a return statement at all

Functions that return values can be used like any other variable

```
print(intFun())
```

```
l = listFun()
l.append(4)
```

From how's to why's

- So far we have covered basics
  - General topics that are common across programming languages:
    - Control flow
    - Parameters/Scope
    - Return values
- Today we will focus more on
  - Extra features
  - When to use them

Modules

- Modules provide a means using functions others have defined
- A module is essentially a collection of functions within a single file
  - Usually related in some way; a common "theme"
- We've seen this before:
  - `random`
  - `math`
  - `urllib`

What happens

mylib.py

```
def head(values):
    return values[0]

def tail(values):
    return value[-1]
```

`import` finds and opens the file, then notes the definitions in that file

The functions are now available in my *namespace*

list_fun.py

```
import mylib


lst = [1, 2, 3]
h = mylib.head(lst)
t = mylib.tail(lst)
```

Documentation

- Python has two (primary) methods of documenting your code
  1. "hashtags"
  2. docstrings
- So far we have discussed the hash method

Hash version

```
# This is my new function

def myFun():
    print('Hello')


# I'll call it here

myFun()
```

Docstring version

```
"""
This is my new function
"""
def myFun():
    print('Hello')

"""
I'll call it here
"""
myFun()
```

Triple quotes denote the beginning and end

Docstrings: comments with benefits

- Hash lines are completely ignored by the interpreter
- Docstrings aren't… completely

```
"""
This module contains
pure awesomeness
"""
def myFun():
    """
    myFun is the bomb!
    """
    return 'bomb'
```

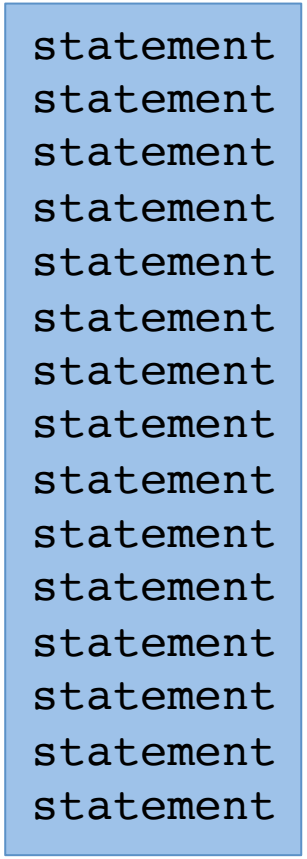
```
>>> import awesome
>>> help(awesome)
>>> help(awesome.myfun)
```

Functions

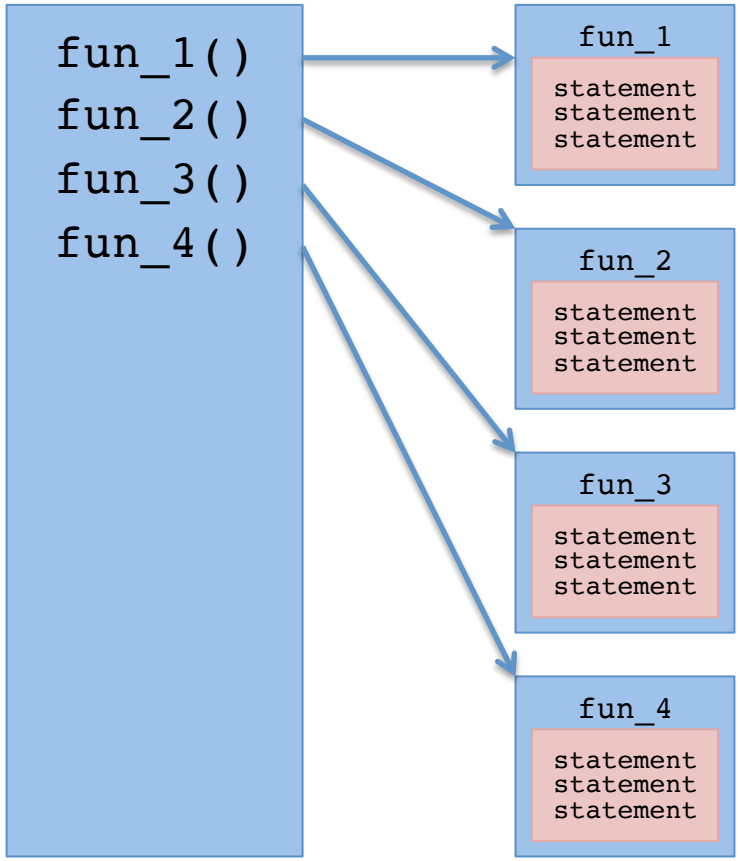
- A sequence of statements that has a name
- Can think of it as a sub-program

Moving away from sequential

So far our programs have been sequential



We will now start to break them up into several "mini" programs



Motivating functions

Functions

1. Provide a name for a group of statements
2. Help to eliminate repetitive code
3. Allow for program development/debugging in stages
4. Provide a means of code reuse (and sharing!)

Naming collections of statements

This is scary

```
guess = input(‘Coordinate 1: ‘)
guess = guess.split(',')
g1 = []
for i in guess:
    g1.append(int(i))

guess = input(‘Coordinate 2: ‘)
guess = guess.split(',')
g2 = []
for i in guess:
    g2.append(int(i))

if master[g1[0]][g1[1]] ==
   master[g2[0]][g2[1]]:
    player[g1[0]][g1[1]] =
    master[g1[0]][g1[1]]
    player[g2[0]][g2[1]] =
    master[g2[0]][g2[1]]
```

Ah, that’s better

```
guess_1 = get_guess(1)
guess_2 = get_guess(2)

if match(master, guess1, guess2):
    update(player, master, guess_1)
    update(player, master, guess_2)
```

Eliminate repetitive code

If you're copy/pasting…

```
l = []
for i in range(100):
    l.append(random.randint(0,10)

m = []
for i in range(100):
    m.append(random.randint(0,10)

# find the average
avg_l = sum(l) / len(l)
avg_m = sum(m) / len(m)
```

… it's time to make a function

```
def random_list(elements):
    l = []
    for i in range(elements):
        r = random.randint(0, 10)
        l.append(r)
    return l

def average(values):
    return sum(values)/len(values)

average(random_list(100))
```

Staged development

Broken? Finished? OMG WTF

```
# 1
val = input('Enter number')
val + 1

# 2
while True:
    nm = input('Name: ')
    d[nm] = input('Number: ')

# 3
for i in range(10000):
    for j in 10000:
        count.append(i*j)
```

Order: restored

```
def problem_1():
    """increment input"""
    val = input('Enter number')
    val + 1

def problem_2():
    """phone book"""
    while True:
        nm= input('Name: ')
        d[nm] = input('Number: ')

def problem_3():
    """aggregate"""
    for i in range(10000):
        for j in 10000:
            count.append(i*j)
problem_3()
```

Code reuse

Printing a board is so hard ☹

```
for row in board:
    print('+---' * len(row) + '+')
    print('|', end='')
    for element in row:
        print(' ', element, ' |',
              end='')
    print()
    print('+---' * len(row) + '+')
```

Not anymore ☺

```
print_board(board)
```

Motivating functions

Functions

1. Provide a name for a group of statements
2. Help to eliminate repetitive code
3. Allow for program development/debugging in stages
4. Provide a means of code reuse (and sharing!)