Intro to Computer Science

Previous

- File I/O

Readings

Gaddis	• Chapter 6

Next

- Refresh
- Functions

Readings

Gaddis	• Chapter 5

Functions

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

- We've seen (used) this before:
  - Built-ins: `print`, `len`
  - Method: `.upper`, `.sort`
  - Modules: `random.randint`
- What we know
  - Functions have a name
  - Functions take parameters
  - Functions can return a value

Functions in Python

def is a keyword: we’re *defining* the function

The function name can be anything you want; standard variable naming applies

```
def fname(arg0, arg1, …, argN):
    block
```

The colon and the block: they just keep on coming back

```
def fname():
    block
```

- The function can have as many arguments as you want;
- or non at all

Our journey through functions

What happens

1. The calling program is suspended
2. The values of the local parameters are assigned to the function parameters
3. The body of the function is executed
4. Control returns to the (next) position in the calling program

What we'll talk about

Pre execution

- Definition and (versus) evaluation

Execution

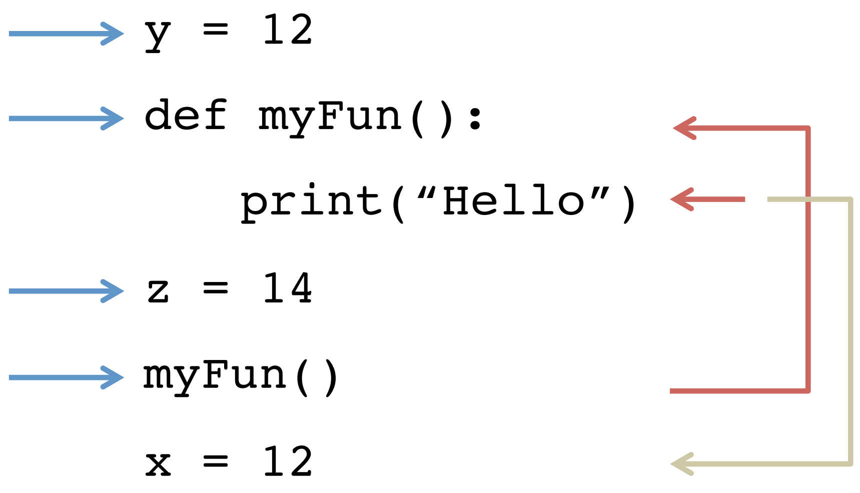
- Variables/scope
- Return values

Post execution

- That just happened!
- Where do we go next?

Definition versus execution

- By default, the interpreter moves from top to bottom
- When it sees a function definition, it makes a note and continues running, *ignoring the block*
- Once that function is called, the block is *executed*
- After execution, *control* returns to the main program



```
y = 12
def myFun():
        print(“Hello”)
z = 14
myFun()
x = 12
```

Functions must be defined to be used

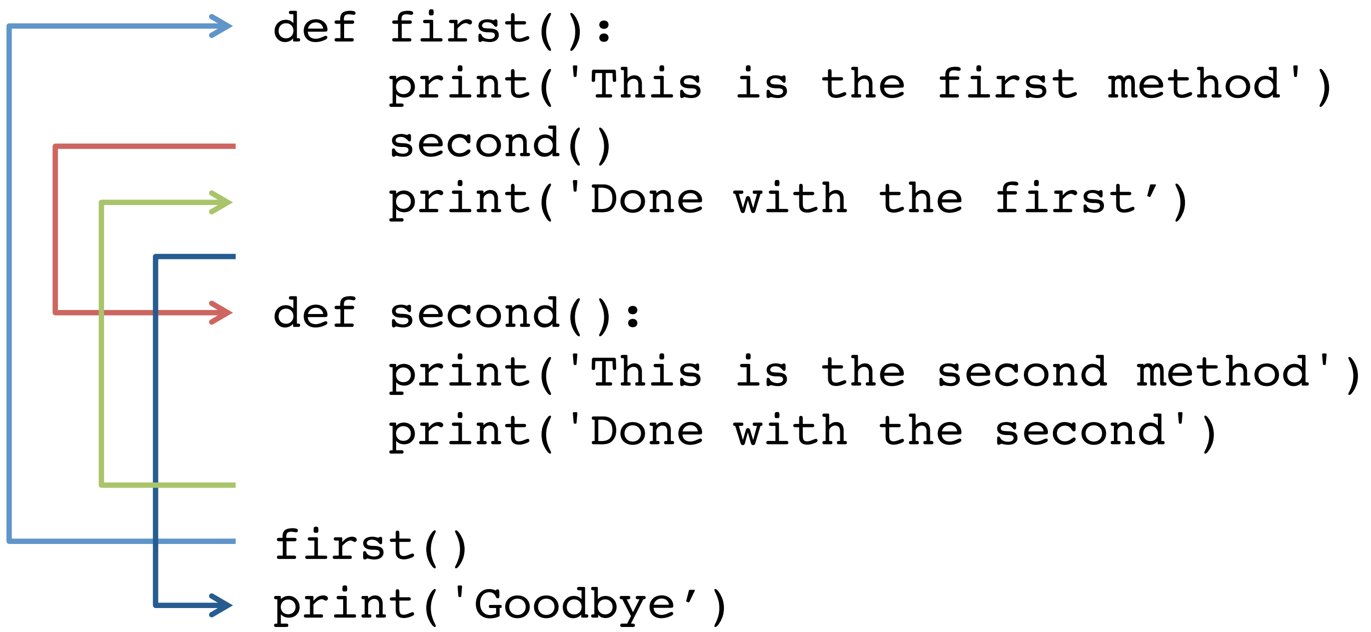
- When it sees a function definition, it makes a note and continues running
- Thus, a function must be defined before it can be *executed*!

```
✗  myFun()

   def myFun():

       print(“Hello”)

✔  myFun()
```

Control flow

A function is like a mini-program:

- You can use all of our previous constructs within the function
- And of course you can call other functions



```
def first():
    print('This is the first method')
    second()
    print('Done with the first’)

def second():
    print('This is the second method')
    print('Done with the second')

first()
print('Goodbye’)
```

Scope

- Recall *scope*: the area of a program where a variable may be referenced
- Scope is generally distinguished between
  - **global**: variables that can be accessed throughout the program
  - **local**: variables can be accessed only within specific parts of the program
- Thus far all of our variables have been global
- Parameters and variables inside the function are local
  - Variables are distinct from variables of the same name elsewhere in the program

Global versus local

Global variables

```
x = 10
def myFun():
   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

Parameters

- Data is passed into functions via parameters
- Recall our original definition:

```
def function(arg1, arg2, arg3):
```

- These arguments (variables) take on the given value within the function

```
def myFun(arg1, arg2, arg3):
    x = arg1
    print(arg2)
```

$\longrightarrow$ Variables are declared in the definition

$\}$ They can be treated like any other variables within the function body

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!

Parameter passing

(This is important. And confusing)

- Parameters are generally passed either by reference, or by value
  - **Reference**: a *reference* to the variable is passed
  - **Value**: a *copy* of the variable is passed
- This matters:
  - Performance implications
  - Data type semantic implications
  - ➢ *Variable value implications*

Value versus reference in general*

By value

```
def myFun(x):
    x += 1

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

$\longrightarrow$ 10

x is updated within the function, but retains its value outside of the function

By reference

```
def myFun(x):
    x += 1

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

$\longrightarrow$ 11

Although x is updated within the function, it's changes are seen outside of the function

*Code on this slide should be taken as an example of the concepts, not necessarily Python

Value versus reference in Python

- In some programming languages, passing by value and passing by reference are explicit
- Not in Python
  - Python uses a hybrid
  - Requires an understanding of the language
  - It’s easy to understand if you understand mutability

Value versus reference in Python

Python copy

- If you update where the variable points, Python makes a copy

```
def myFunc(a, b):
    a = 10
    b = 12

x = 4
y = 5
myFunc(x, y)
```

x and y are unchanged to the callller

Python reference

- If you update the value of the variable, Python uses the reference
- *This only matters for mutable data types!*

```
def myFunc(x, y):
    x.append(y)


a = [1, 2, 3]
myFunc(a, 4)
```

a is now has an additional value

Return value

- Data is passed out of functions via return values
- Functions that do not return a value are called void functions

void functions	• `print` • most list methods
non-void functions	• `input` • `len` • `range` • most string methods

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]`

Functions that return values can be used like any other variable

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

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

Return is like break!

- Return can be used anywhere within a function
- Once is appears, the function is done!

```
def myFun():
    for i in range(10):
        if i == 5:
            break
    print('Got here')

myFun()
```

```
def myFun():
    for i in range(10):
        if i == 5:
            return
    print('Got here')

myFun()
```

Void return is the default

- In Python, if a function has no explicit return, it returns nothing by default

```
def myFun():
    for i in range(10):
        print(i)
    return

myFun()
```

```
def myFun():
    for i in range(10):
        print(i)


myFun()
```

These functions are the same!