Today

- Arrays and Functions wrap-up
- Naming scope

Naming scope

```
def f(x):
    y = 1
    return x + y
print(f(2))
print(y)
```

What happens?

```
def f(x):
    y = 1
    return x + y
y = 2
print(f(3))
print(y)
```

```python
def f(x):
    y = 1
    return x + y
y = 2
print(f(3))
print(y)
```

Current line	Why we went to this line	Variables after current line runs	Output so far
y = 2	It was the first line (after the definition of the function)	y = 2	
`print(f(3))`	It was the next line	x = 3 y = 2	
Call to the function f with argument value 3			
`y = 1`	We just called the function f, so we go to its first line	x = 3 y = 1	
`return x + y`	It was the next line	x = 3 y = 1	
f ends now with a return value of 4			
`print(f(3))`	We now return to the line that called the function with the return value of the function	y = 2 (The function is over, so any changes it made to any variables are discarded and any variables that existed before the function call retain their previous values	4
`print(y)`	It was the next line	y = 2	4 2

Basic Rule

- Variables are “local” to function they are declared in
- Like all variables, they begin their life undefined
- When function ends they become inaccessible
- Variables with same name in two different functions or the “main” program have their own “lives”

Scope →ISA

- Each variable gets its own register
- Before calling a function we save all registers (variables)
- After function “returns” we restore all registers (variables)
- Save to a place called “stack”

Why have Functions?

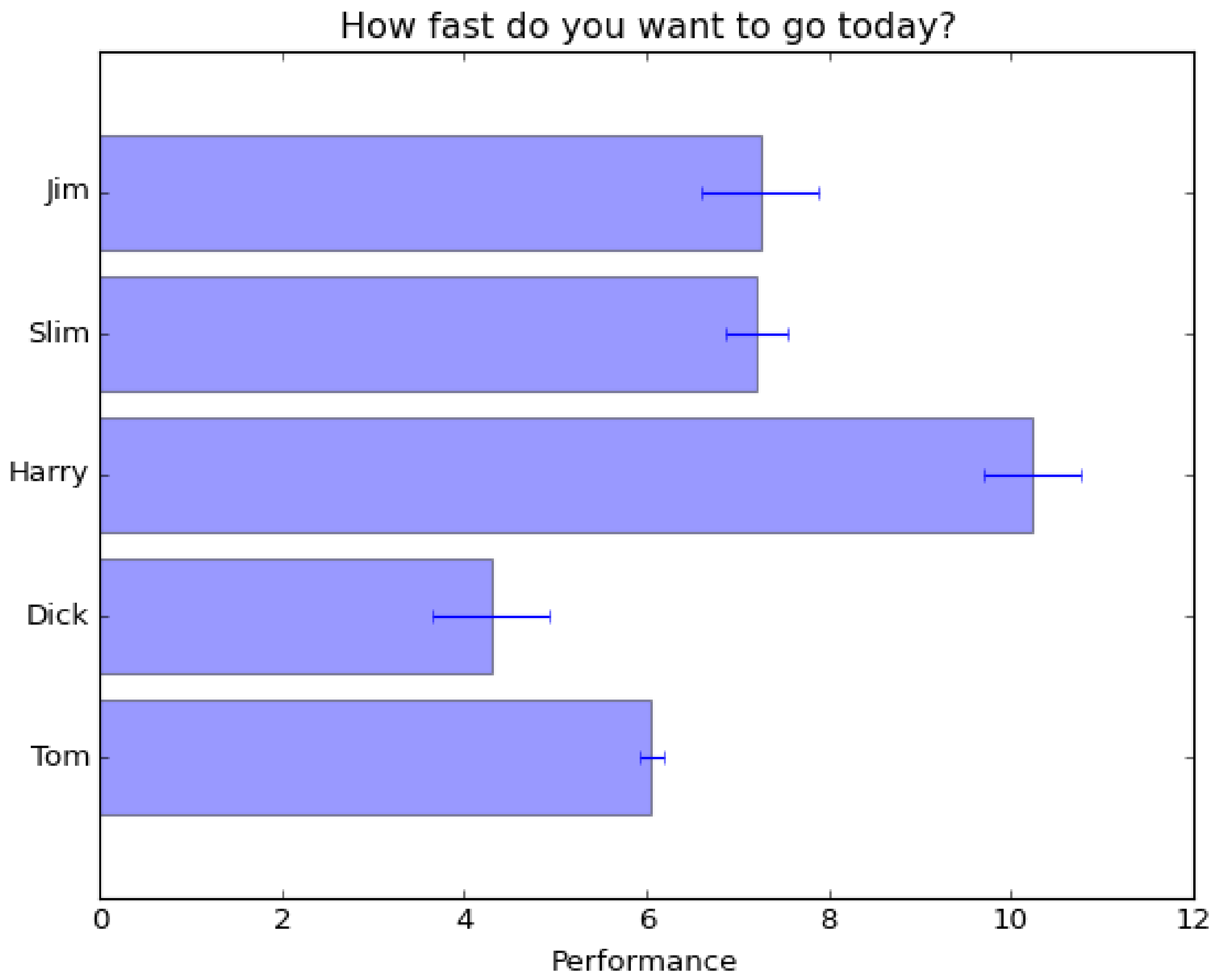
- Readability
- Maintainability
- Modularity/composition

Libraries

- Collection of functions is called a library
- Many examples:
  - Googlemapsapi, graphing library, statistics libraries…

```
from googlemaps import Client
api_key = "AIzaSyADna65ndIZRBBvx-V213ZLqHxO5KMyApY"
gmaps = Client(api_key)
origin = "1210 W. Dayton St., Madison, WI"
destination = "1605 Linden Dr, Madison, WI"
d = gmaps.directions(origin, destination)

print d.distance

for x in d.steps
    print x['html_instructions']
```

```
import matplotlib.pyplot as plt; plt.rcdefaults()
import numpy as np
import matplotlib.pyplot as plt

# Example data
people = ('Tom', 'Dick', 'Harry', 'Slim', 'Jim')
y_pos = np.arange(len(people))
performance = 3 + 10 * np.random.rand(len(people))
error = np.random.rand(len(people))

plt.barh(y_pos, performance, xerr=error, align='center',
alpha=0.4)
plt.yticks(y_pos, people)
plt.xlabel('Performance')
plt.title('How fast do you want to go today?')

plt.show()
```


How fast do you want to go today?
Jim
Slim
Harry
Dick
Tom
0
2
4
6
8
10
12
Performance

Summary

- Arrays & Functions
- Naming scope
- Variables

CE/ECE 252 Lecture 8: Arrays and More

2015 Sept 21

Transcribed Notes

Function Example 1

```
def f(x)
    y = 1
    return(x + y)
print(f(21))
print(y)            # y is not defined at this point,
                    # y above is out of scope
```

Function Example 2

```
def f(x)
    y = 1
    return(x + y)
y = 2
print(f(3))
print(y)            # will print 2,
                    # y in G(x) is out of scope
```

Scoping

All variables are local to the function they are defined in. They cease to exist outside of the function.

Global variables are accessible from anywhere, but try to avoid them.

Scope -> ISA

```
def f(x)
    y = 1               # ld R14, 1
    z = f3(...)         # need a stack of return addresses
    return(x + y)       # add R15, R4, R14
                        ~~# branch +3~~
# branch +3 will not work if called from multiple locations
# we need to store the return address when calling
# return, jump back to return address
y = 2                   # ld R13, 2
print(f(3))             ~~# branch -3~~
                        # ld R4, 3
                        # callWithArgument -4, R4
                        # jump up 4 lines of code
                        # send R4 as the argument
print(y)
print(f(75))

def f2(x, y, z, a, b, c, d)   # callWithArgument -8, ...
    return (x + y + z + a + b + c + d)
# problem: all instructions can have at most 4 numbers
# solution:
#   create an array,
#   put values into the array,
#   pass in the location of the array
```

Arrays

```
arr[] = {1, 4, 10, 45}     # array starts at R27
x = arr[0]                 # ldarray R11, R27, 0
                           # load into R11
                           # R27 is where the array begins
                           # 0 is the offset
y = arr[2]                 # ldarry R14, R27, 2
# alternatively: add R27, R27, 2; ldarray R14, R27, 0
z = 73                     # ld R13, 73
```

Register File	**Memory**
	Location 173 (arbitrary) 1
	4
	10
	45
Register 27 (arbitary) 173	
No implication	

Arrays and functions are simple programming constructs.