More on Functions

Thomas Schwarz, SJ
Marquette University

Functions of Functions

- Functions are full-fledged objects in Python
  - This means you can pass functions as parameters
  - Example: Calculate the average of the values of a function at -n, -n+1, -n+2, …, -2, -1, 0, 1, 2, … , n-2, n-1, n
    - The function needs to be a function of one integer variable
- Example:
  - $n = 2$, function is squaring
  - Return value is $((-2)^2 + (-1)^2 + 0^2 + 1^2 + 2^2)/5 = 2$

Functions of Functions

- We first define the averaging function with two arguments
  - The number *n*
  - The function over which we average, called `func`

```
def averaging(n, func):
```

Functions of Functions

- Inside the function, we create an accumulator and a loop index, running from -*n* to *n.*

```
def averaging(n, func):
    accu = 0
    for i in range(-n, n+1):
```

Functions of Functions

- Inside the loop, we modify the accumulator `accu` by adding the value of the function at the loop variable.

```
def averaging(n, func):
    accu = 0
    for i in range(-n, n+1):
        accu += func(i)
```

Functions of Functions

- There are 2*n*+1 points at which we evaluate the function.
- We then return the average as the accumulator over the number of points

```
def averaging(n, func):
    accu = 0
    for i in range(-n, n+1):
        accu += func(i)
    return accu/(2*n+1)
```

Functions of Functions

- In order to try this out, we need to use a function
- We can just define one in order to try out our averaging function

```
def square(number):
    return number*number

def averaging(n, func):
    accu = 0
    for i in range(-n, n+1):
        accu += func(i)
return accu/(2*n+1)

print(averaging(2, square))
```

Local and Global Variables

- A Python function is an independent part of a program
  - It has its own set of variables
    - Called local variables
  - It can also access variables of the environment in which the function is called.
    - These are global variables
- The space where variables live is called their scope
- We will revisit this issue in the future

Examples

```
a=3
b=2
def foo(x):
    return a+x
def bar(x):
    b=1
    return b+x

print(foo(3), bar(3))
```

- a and b are two global variables
- In function foo:
  - *a* is global, its value remains 3
- In function bar:
  - *b* is local, since it is redefined to be 1

Preview of Scoping: The global keyword

- In the previous example, we generated a local variable *b* by just assigning a value to it.
- There are now two variables with name *b*
- In bar, the global variable is hidden
- If we want to assign to the global variable, then we can use the keyword global to make *b* refer to the global variable. An assignment then does not create a new local variable, but rather changes the value of the old one.

Example

```
a = 1
b = 2

def foo():
    global a
    a = 2
    b = 3
    print("In foo:" , "a=", a, " b=", b)

print("Outside foo: " ,"a=", a, " b=", b)
foo()
print("Outside foo: " ,"a=", a, " b=", b)

##Outside foo:  a= 1  b= 2
##In foo: a= 2  b= 3
##Outside foo:  a= 2  b= 2
```

- In foo:
  - A local variable *b*
  - A global variable *a*
  - The value of *a* changes by executing *foo*( )

Scoping

- Scoping is definitely an advanced topic
  - The take-home is:
    - ***<u>Don't ever, ever use global variables</u>***
      - ***<u>Unless you really need to</u>***.
- Under most circumstances, you should pass variables as arguments.
  - **<u>Python Philosophy:</u>** ***Rules are followed by convention, there is no enforcement***
    - Because sometimes you need to make exceptions