

# Functions



Global Knowledge®

# Objectives

- See how to define functions and return values
- Understand the variable scope rules of Python
- Treat functions as objects
- See how to define functions in any order
- Pass a variety of arguments to functions
- Define inline functions with lambdas
- Create light-weight classes via closures

# Defining functions

- Functions are defined using the **def** keyword
  - can accept **arguments** (do not state the type)
  - can **return** values (do not state the type)
  - can be stand-alone or belong to classes (same syntax)

```
def somemethod(setSize):  
    r = random.randint(0, 10000)  
    if r % setSize == 0:  
        return "{0} is on the edge of {1}" \  
            .format(r, setSize)  
    else:  
        return "{0} is in the middle of {1}" \  
            .format(r, setSize)
```

```
print( somemethod( 4 ) )  
# prints: 5396 is on the edge of 4
```

# Functions [return values]

- Functions can **return** values or be 'void' methods

```
def updateuser(user): # a void or None method
    db.update(user)
    # implicit: return None
```

```
def findEmail(userId): # a string return type
    user = db.find(userId)
    return user.email
```

- They can even return **multiple values with tuples**

```
# returns (string, string) tuple
def findUserInfo(userId):
    user = db.find(userId)
    return user.email, user.name
```

```
email = findEmail(42)
email, userName = findUserInfo(42)
```

# Functions [variable scope]

- Variable scope is based on initialization
  - Variables **initialized** within a function are scoped to that function
  - Variables first **used** within functions are global

```
def scopeMethod():  
    inner = 7  
    print(inner) # local, prints 7  
  
print(inner) # NameError
```

```
outer = 6  
  
def scopeMethod():  
    print(outer) # global, prints 6
```

# Functions [as objects]

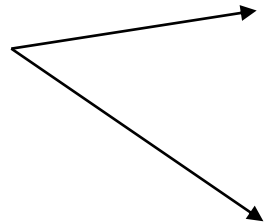
- Functions can be treated as objects
  - can be passed around
  - can hold values: **someMethod.key = value**

```
def strategyMethod(num, predicate):  
    if predicate(num):  
        print('would perform action')  
    else:  
        print('not happening!')  
  
def isByThree(num):  
    return num % 3 == 0  
  
strategyMethod(6, isByThree)  
  
# prints: would perform action
```

# Functions [order of definition]

- Methods can be defined in any order provided
  - All methods are defined at the time of invocation

m3 is defined  
after m1 uses it



Use `__name__`  
convention to invoke  
code at the very  
bottom of your file

```
def m2():  
    print("M2")  
  
def m1():  
    print("M1")  
    m2()  
    m3()  
  
# m1() <-- no, would fail (m3 missing)  
  
def m3():  
    print("M3")  
  
# m1() <-- no, would execute on import  
  
if __name__ == "__main__":  
    m1()
```

# Function [arguments]

- Functions have a lot of flexibility to accept arguments
  - positional arguments (default)
  - default values for keyword arguments

```
def positional(x, y, z=0):  
    print(x, y, z)  
  
positional(1, 2, 6)    # prints 1, 2, 6  
positional(1, 2, z=6)  # prints 1, 2, 6  
positional(1, 2)       # prints 1, 2, 0
```



# Function [\*args]

- Functions can take additional, variable length arguments
  - passed as a tuple
  - Indicated with **\*args**
  - the args name is just a convention

```
def positional(x, y, *args):  
    print(x, y, args)
```

```
positional(1, 2, 6, 7, 8) # prints 1 2 (6, 7, 8)
```

# Function **\*\*kwargs**

- Functions can take additional, named arguments
  - called **keyword arguments**
  - passed as a dictionary
  - indicated with **\*\*kwargs**
  - the kwargs name is just a convention

```
def keywordArguments(x, y, **kwargs):  
    print(x, y, kwargs)  
  
keywordArguments(1, 2, z=2, u=1, mode="reversed")  
# prints 1 2 {'u': 1, 'z': 2, 'mode': 'reversed'}  
  
# you can even 'project' a dictionary as kwargs  
dargs = {'u': 2, 'mode': 'forward', 'z': 7}  
keywordArguments(1, 4, **dargs)  
# prints 1 4 {'u': 2, 'mode': 'forward', 'z': 7}
```

## Function [closures]

- Function closures can serve as light-weight classes
- They bundle
  - data
  - behavior
  - encapsulation

# Function [closure definition]

Closure method is created and returned. (`nonlocal` captures state)



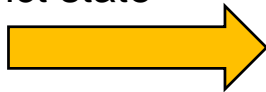
```
def createCounter(startingValue):  
    def counterClosure():  
        incValue = 2  
        nonlocal startingValue  
        startingValue += incValue  
        return startingValue  
  
    return counterClosure
```

Two instances of the closure are created and held.



```
counter1 = createCounter(5)  
counter2 = createCounter(10)
```

They use private, shared behavior but distinct state



```
counter1() # returns 7  
counter1() # returns 9  
counter2() # returns 12  
counter1() # returns 11
```

# Lambdas

- Lambdas are short, anonymous functions
- Use the form: `lambda args: expression`

```
f = lambda x: x * x  
f(7) # 49
```

- can have multiple arguments

```
f = lambda x, y: x + y  
f(7, 11) # 18
```

- can have optional arguments

```
f = lambda x=11: x * x  
f(7) # 49  
f() # 121
```

# Lambdas [as arguments to methods]

- The real value of lambdas are as **arguments**

```
def strategyMethod(num, predicate):  
    if predicate(num):  
        print('would perform action')  
    else:  
        print('not happening!')
```

```
strategyMethod(6, lambda x: x % 3 == 0)  
# prints would perform action
```

```
# state capture and closures are simpler with lambdas  
modSize = 5  
strategyMethod(6, lambda x: x % modSize == 0)  
# prints not happening!
```

# Lambdas [as arguments to library methods]

- The real value of lambdas are as **arguments**
  - This especially true for library methods

```
nums = [1, -2, 4, -3, 5]

nums.sort(key = lambda x: -abs(x))

print(nums) # prints [5, 4, -3, -2, 1]
```

# Summary

- Functions are the scoping mechanism in Python
- Functions are first class objects
- Use the `__name__` convention to execute functions at the right time
- Functions take positional, optional, additional, and keyword arguments
- Closures allow bundling of data, encapsulation, and behavior into a single method
- Lambdas serve as inline, concise method definitions