CME 211: Lecture 4: Functions and Complexity Analysis

Python functions

- Code we have seen so far has been executed in linear fashion from top to bottom, sometimes repeating one or more lines in a loop body
- Functions allow us to:
- Replace duplicated code with one centralized implementation within a single program
- Reuse code across different programs
- Easily use code developed by others
- Develop, test, debug code in isolation from other code
- Analogous to mathematical functions

Defining a function in Python

Let's start with an example:

```
def print_hello(name):
    print("Hello, {}".format(name))
```

If Python encounters a function name without parens `()`, it tells us that it is a function:

```
print_hello
```

Call the function:

```
print_hello("CME211")
```

Anatomy of a Python function

```
def function_name(input_argument):
    # function body
    print("you guys rock")
```

1. start with `def` keyword
2. give the function name
3. followed by comma separated list of input arguments, surrounded by parentheses
- just use `()` for no input arguments
4. followed by a colon `:`
5. followed by **indented** function body

Return a value

Use the `return` keyword to return an object from a function:

```
def summation(a, b):
    total = 0
    for n in range(a,b+1):
```

```
        total += n
    return total

c = summation(1, 100)
c
```

Return multiple values

Separate multiple return values with a comma:

```
def sum_and_prod(a,b):
   total = 0
   prod = 1
   for n in range(a,b+1):
       total += n
       prod *= n
   return total, prod
```

Call the function:

```
a = sum_and_prod(1,10)
print("a:", a)
print("type(a):", type(a))
```

The `return` keyword packs multiple outputs into a tuple. You can use Python's tuple unpacking to nicely get the return values in calling code.

```
a, b = sum_and_prod(1,10)
print("a:", a)
print("b:", b)
```

Variable scope

Let's look at an example to start discussing variable scope:

```
total = 42
def summation(a, b):
    total = 0
    for n in range(a, b+1):
        total += n
    return total

a = summation(1, 100)
a

print("total:", total)

print("n:", n)
```

Function bodies have a local namespace. In this example the `summation` function does not see the variable `total` from the top level scope. `summation` creates its own variable `total` which is different! The top level scope also cannot see variables used inside of `summation`.

Reference before assignment to a global scope variable will cause an error:

```
total = 0
def summation(a, b):
    for n in range(a, b+1):
```

```
        total = total + n
        #           ^
        #       reference before assignment
    return total

a = summation(1, 100)
```

Variable scope examples

It is possible to use a variable from a higher scope. This is generally considered bad practice:

```
a = ['hi', 'bye']
def func():
    print(a)

func()
```

Even worse practice is modifying a mutable object from a higher scope:

```
a = ['hi', 'bye']
def func():
    a.append('hello')

func()
print(a)
```

Python will not let you redirect an identifier at a global scope. Here the function body has its own `a`:

```
a = ['hi', 'bye']
def func():
    a = 2

func()
print(a)
```

Accessing a global variable

This is bad practice, do not do this. We will take off points. We show you in case you run into it.

```
total = 0
def summation(a,b):
    global total
    for n in range(a, b+1):
        total += n

a = summation(1,100)
print("total:",total)
```

Functions must be defined before they are used

Functions must be defined before they are used! See the file `order1.py`:

```
def before():
    print("I am function defined before use.")
```

```
before()
after()

def after():
    print("I am function defined after use.")
```

Output:

```
$ python3 order1.py
I am function defined before use.
Traceback (most recent call last):
  File "order.py", line 5, in <module>
    after()
NameError: name 'after' is not defined
$
```

A function may refer to another function defined later in the file. The rule is that functions must be defined before they are actually invoked/called.

See `order2.py`:

```
def sumofsquares(a, b):
    total = 0
    for n in range(a, b+1):
        total += squared(n)
    return total

def squared(n):
    return n*n

print(sumofsquares(1,10))
```

Output:

```
$ python3 order2.py
385
```

Passing convention

Python uses pass by object reference. Python functions can change mutable objects referred to by input variables

```
def do_chores(a):
    a.pop()

b = ['feed dog', 'wash dishes']
do_chores(b)
print(b)
```

`int`s, `float`s, and `str`ings are immutable objects and cannot be changed by a function:

```
def increment(a):
    a = a + 1

b = 2
increment(b)
b
```

Pass by object reference

- Python uses what is sometimes called pass by object reference when calling functions
- If the reference is to a mutable object (e.g. lists, dictionaries, etc.), that object might be modified upon return from the function
- For references to immutable objects (e.g. numbers, strings), by definition the original object being referenced cannot be modified

Default and keyword arguments

We have seen that the behavior of some Python functions can be modified by passing keyword arguments. Keyword arguments have default values. For example, the `print` function has optional `end` and `sep` arguments:

```
print("first line, ")
print("second line")
```

```
print("first line, ", end="")
print("second line")
```

```
print(1,2,3,4,5,6,7)
print(1,2,3,4,5,6,7, sep=", ")
```

It is simple to use this feature when defining functions:

```
def func(x, a = 1):
    return x + a

print("   func(1) =", func(1))
print("func(1, 2) =", func(1, 2))
```

The default value is used if the argument is not specified when the function is called.

Multiple default arguments

Consider the function prototype: `func(x, a=1, b=2)`.

Suppose we want to use the default value for `a`, but change `b`:

```
def func(x, a=1, b=3):
    return x + a - b

print("     func(2) =", func(2))
print("  func(5, 2) =", func(5, 2))
print("func(3, b=0) =", func(3, b=0))
```

Keyword arguments may be passed in any order:

```
func(10, b=5, a=7)
```

See the Python Tutorial section on defining functions for more info.

Docstring

It is important that others, including *you-in-3-months-time* are able to understand what your code does.

This can be easily done using a so called "docstring", as follows:

```
def nothing():
    """ This function doesn't do anything. """
    pass
```

We can then read the docstring from the interpreter using:

```
help(nothing)
```

Built-in Python functions also have documentation, see `help(print)`:

```
help(print)
```

Functions as objects

In Python everything is an object, this includes functions. It is possible to pass functions to other functions:

```
def simple_function():
    print("hello from simple_function()")

def function_caller(f):
    # just call the function f
    f()
```

Now, we can pass `simple_function` to `function_caller`:

```
function_caller(simple_function)
```

This is useful when combined with Python's `map` and [`filter`][py-filetr] functions.

`map`

The `map` function takes a mapping function and one or more iterable objects as input arguments. The `map` returns an iterator that applies the input function to every item of the iterable object yielding the results. Take for example square function

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

and a list as `map` inputs:

```
map(square, [1,2,3,4,5,6])
```

The return value is an iterator can be used in a `for` loop:

```
for s in map(square, [1,2,3,4,5,6]):
    print(s,end=', ')
print()
```

The iterator does not create a new list, it simply calculates all mapped values. If we want to create a list with mapped values, we pass the iterator as the input argument of the list constructor:

```
list(map(square,[1,2,3,4,5,6]))
```

In Python 2.x, the `map` function is returning a container with mapped data. In Python 3.x, it is programmers responsibility to decide if and when mapped data should be stored in memory.

`filter`

The `filter` function returns an iterator over items in a container for which the input function returns `True`:

```
def isodd(x):
    return x % 2 != 0

list(filter(isodd,[1,2,3,4,5,6,7]))
```

Lambda functions

A **lambda** function is simply a function without a name. These are also called **anonymous** functions.

They are used as an alternative way to define short functions:

```
cube = lambda x: x*x*x
print("cube(3) = ", cube(3))

list(map(lambda x: x*x*x, [1,2,3,4,5,6,7,8,9]))
```

Example of a bad function

```
def add(a, b):
    # I wrote this function because Nick
    # is mean and is making us write three functions in a homework
    return a + b
```

Recommended Reading

From **Learning Python, Fifth Edition** by Mark Lutz

- Chapter 6: The Dynamic Typing Interlude (i.e. references and objects)
- Chapter 16: Function Basics
- Chapter 17: Scopes
- Chapter 18: Arguments