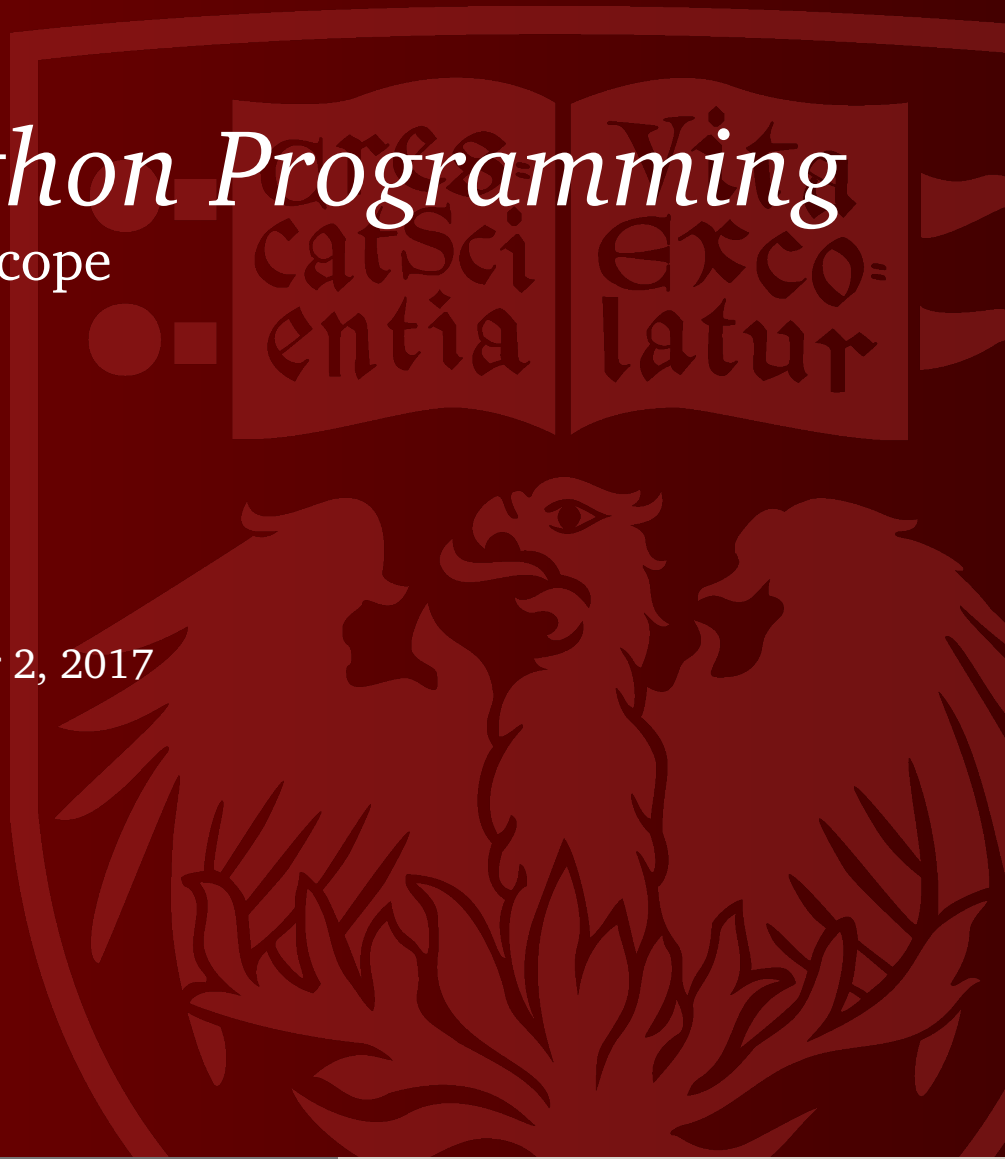
MPCS 51042-2: *Python Programming*

Week 2: Functions, generators, and scope



October 2, 2017

Code Review

Functions

Functions

- Set of statements that can be called more than once
- In CS literature, often called subroutines, procedures, callable units, etc.
- Allows programmer to avoid copy/paste to repeat same task, maximizing code reuse

Functions

```
def name(arg1, arg2, ..., argN):
    statements
    ...
    return value
```

- Defines a new function called `name`
- **`return`** can appear anywhere in body of function
- **`return`** can also be omitted—dropping off a function is equivalent to returning **`None`**
- **`def`** is an executable statement: when it runs, it creates a new function object and assigns it to a name
- Statements inside function don't run until function is called

Function calls

```
>>> def add(x, y):
...     return x + y
...
>>> add(3, 5)
8
>>> x = add(1, 2)
>>> x
3
>>> add("Python", " programming")
'Python programming'
```

Local variables

Local variables only exist inside the function

```
def intersect(seq1, seq2):
    common_set = set()
    for x in seq1:
        if x in seq2:
            common_set.add(x)
    return common_set
```

- Although `common_set` was defined locally, it is returned and might be bound to a new identifier
- The `x` variable on the other hand is local to the function

Function arguments

- Arguments are passed by assigning objects to local variables
- Assigning to argument names does not affect the caller

```
>>> x = 5
>>> def foo(x):
...         x = 10
...
>>> x
5
```

- Changing a mutable object may impact caller

```
>>> x = list(range(5))
>>> def append(seq, item):
...         seq.append(item)
...
>>> append(x, 10)
>>> x
[0, 1, 2, 3, 4, 10]
```

Argument matching: function

Considerable flexibility in how we 1) define arguments and 2) pass arguments

- Required arguments

```
def f(a, b, c):
    print(a, b, c)
```

- Optional arguments (default provided)

```
def f(a, b=1, c='spam'):
    print(a, b, c)
```

Argument matching: caller

The caller can provide arguments by position or by keyword:

```
>>> def f(a, b=1, c='spam'):
...     print(a, b, c)
...
>>> f(4, 4, 4)
4 4 4
>>> f(4, 4)
4 4 spam
>>> f(4)
4 1 spam
>>> f(4, c='Ni')
4 1 Ni
```

Collecting positional arguments

- Function that takes any number of arguments:

```
def reduce_add(*args):
    s = 0
    for x in args:
        s += x
    return s
```

- Can combine normal positional arguments with collected arguments

Collecting keyword arguments

Using `**` gives us a dictionary of keyword arguments

```
>>> def make_dict(**kwargs):
...     for key, value in kwargs.items():
...         print(f'{key} is {value}')
...
>>> make_dict(name='John', course='Python', age=25)
name is John
course is Python
age is 25
```

Unpacking arguments

The caller can also use `*` and `**` to unpack iterables into positional or keyword arguments

```
>>> def double(x, y, z):
...     return 2*x, 2*y, 2*z
...
>>> point = (3, -2, 7)
>>> double(*point)
(6, -4, 14)
>>> point = {'x': 0, 'y': 3, 'z': 4}
>>> double(**point)
(0, 6, 8)
```

Argument matching summary

Syntax	Meaning
`f(name)`	Matched by position
`f(name=value)`	Matched by name
`f(*iterable)`	Pass each object as a positional argument
`f(**dict)`	Pass each key/value pair as a keyword argument
`def f(name)`	Matches any argument by position/name
`def f(name=value)`	Default argument value
`def f(*args)`	Collect remaining position arguments in a tuple
`def f(**kwargs)`	Collect remaining keyword arguments in a dictionary
`def f(*args, name)`	Arguments must be passed by keyword-only

Multiple return values

To return multiple values, simply use a tuple

```
def return_args(x, y, z):
    return (x, y, z)
```

or, because parentheses are optional:

```
def return_args(x, y, z):
    return x, y, z
```

Function overloading

- In some languages, functions can be created with the same name but different *signatures*, e.g., in C:

```
void f();
void f(int x);
void f(int x, double y);
int f(int x);
```

- Python does not allow this—use argument defaults instead

Scope

Namespaces and scope

- When you use a name in a program, Python creates/looks up the name in a *namespace*
- The location of a name's assignment determines the *scope* of its visibility to your code
- Functions create their own namespace
  - Names inside a function (local variables) cannot be seen outside of the `def`
  - Names inside the function do not clash with names outside

How scope is determined

- Names assigned at the top-level (outside of a function) in a file have "global" scope
- Global scope only covers a single file
- Names inside a function are local unless specified otherwise
- There is also a "built-in" namespace with things like `abs()`, `sum()`, `int()`, etc.

Inside functions

- Assignment statements create/change local names
- Referencing a name in an expression searches four scopes:
  1. Local (L) scope of the function
  2. Local scope of any enclosing (E) functions
  3. Global (G) scope of the file
  4. Built-in (B) scope
- This scheme is called the *LEGB* rule

The global statement

- If you want to re-assign/modify a name with global scope inside of a function, use the **`global`** statement

```
>>> x = 2
>>> def f():
...         global x
...         x += 1
>>> f()
>>> x
3
```

Nested functions

- **def** is just an executable statement that binds a name to a function object
- Permissible to place a **def** anywhere a statement is expected

```
>>> def f1():
...     x = 'hello world'
...     def f2():
...         def f3():
...             print(x)
...         f3()
...     f2()
>>> f1()
hello world
```

Closures

- When a function is nested inside another function, it remembers the enclosing scope
- The combination of a function and variables defined in the enclosing scope (nonlocal variables) is called a *closure*

```
def remember():
    history = []
    def f(*args):
        history.append(args)
        return history
    return f
```

- Closures allow functions to store *state*

The nonlocal statement

- Similar to **`global`**, the **`nonlocal`** statement allows us to change variables defined in an enclosing scope

```
def count_calls(func):
    count = 0

    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print(f'Called {count} times')
        return func(*args, **kwargs)

    return inner
```

Advanced topics

First class objects

Functions are "first-class" objects:

- They can be (are) created dynamically
- They can be assigned to other names
- They can be passed to/returned from functions
- Names of functions are treated as ordinary identifiers (can be reassigned)

Recursive functions

- Functions can be called recursively

```
def fib(n):
    if n < 2:
        return n
    return fib(n-2) + fib(n-1)
```

- Can be used for arbitrarily nested structures
- Python doesn't implement tail recursion elimination
- Note: each successive call has its own local scope

Anonymous functions

- Functions generally have a definition (the body) and an identifier (the name)
- It's possible to create *anonymous* functions, i.e., functions with no identifier

```
def add(x, y):
    return x + y

add = lambda x, y: x + y
```

- Note that lambda is an expression—can be used in places where **def** cannot
- The expression to the right of the : is a single statement

Functional programming constructs

- `map`: call function on each of an iterable's items
- `filter`: filter items based on a test function
- `reduce`: combine items in an iterable
  - In Python 3.x, `reduce` got moved to the `functools` module

Comprehensions and Generators

Applying functions

We saw that `map` allows us to apply a function to each item in a sequence. Thus, we can replace

```
>>> y = []
>>> for x in range(10):
...     y.append(2**x)
...
>>> y
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

with a one-liner:

```
>>> list(map(lambda x: 2**x, range(10)))
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

List Comprehensions

A *list comprehension* applies an expression to each item in an iterable

```
>>> [2**x for x in range(10)]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

Nesting and filtering

Comprehensions can be nested, and we can also use `if` to filter items based on a condition, much like `filter`

```
>>> [(x, y) for x in [1, 3, 4] for y in [3, 1, 2] if x != y]
[(1, 3), (1, 2), (3, 1), (3, 2), (4, 3), (4, 1), (4, 2)]
```

This is equivalent to:

```
>>> z = []
>>> for x in [1, 3, 4]:
...     for y in [3, 1, 2]:
...         if x != y:
...             z.append((x, y))
...
>>> z
[(1, 3), (1, 2), (3, 1), (3, 2), (4, 3), (4, 1), (4, 2)]
```

Set comprehensions

The same comprehension syntax can be used to construct sets

```
>>> {x**2 for x in range(10)}
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

which is equivalent to:

```
>>> y = set()
>>> for x in range(10):
...     y.add(x**2)
...
>>> y
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

Dictionary comprehensions

...and dictionaries as well!

```
>>> import os
>>> {f: os.path.abspath(f) for f in os.listdir()
...                         if f.startswith('w')}
{'week1': '/home/romano/mpcs-python/week1',
 'week7': '/home/romano/mpcs-python/week7',
 'week4': '/home/romano/mpcs-python/week4',
 ...}
```

Generators

- **`yield`** statement defines a generator function

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1
```

- Often used to feed iteration

```
for x in countdown(10):
    print(f'T-minus {x}')
```

Understanding generators

- Despite the fact that it uses `def`, a generator is substantially different from a normal function
- Creating a generator does not actually run the body of the generator
- Advancing the generator requires calling `next(g)`
- Generators can provide better performance (e.g., over lists) by:
  - Saving memory since one item of the iterable is produced at a time (*lazy evaluation*)
  - Not requiring that all items be evaluated up front

Generator expressions

- Generator expressions provide lazy evaluation with the same syntax as list comprehension—replace [] with ()

```
>>> [random.random() for i in range(5)]
[0.9238587657477094,
 0.7488805295977542,
 0.20861578965348637,
 0.10608107884180329,
 0.8368172308939684]
>>> g = (random.random() for i in range(5))
>>> g
<generator object <genexpr> at 0x7fd836184200>
>>> next(g)
0.4921472371895316
>>> next(g)
0.2853502211211741 4
```

Generator expressions as function arguments

Often, you will see generator expressions as function arguments, where they don't require an extra set of parentheses

```
>>> ' '.join(s.upper() for s in 'aaa:bbb:ccc'.split(':'))
'AAA BBB CCC'
```

Single use only

Generator functions/expressions can only be used once!

```
>>> g = (c*3 for c in 'python')
>>> list(g)
['ppp', 'yyy', 'ttt', 'hhh', 'ooo', 'nnn']
>>> list(g)
[]
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Assignment/Reading

Suggested Reading

- Learning Python: chapters 16–20
- Fluent Python: chapters 5 and 6