301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

***AP-27**: Python: Functions, Decorators, Namespaces*

Next topics

- Function definition
- Positional and keyword arguments of functions
- Functions as objects
- Higher-order functions and decorators
- Namespaces and Scopes

Functions in Python - Essentials

- Functions are first-class objects
- All functions return some value (possibly **None**)
- Function call creates a new scope
- Parameters are passed by object reference
- Functions can have optional keyword arguments
- Functions can take a variable number of args and kwargs
- Higher-order functions are supported

Function definition (1)

- Positional/keyword/default parameters

```
def sum(n,m):
    """ adds two values """
    return n+m

>>> sum(3,4)
7
>>> sum('hel','lo')
'hello'
>>> sum(m='lo',n='hel') # keyword parameters
'hello'

#-------------------------------------

def sum(n,m=5): # default parameter
    """ adds two values, or increments by 5 """
    return n+m

>>> sum(3)
8
```

Function definition (2)

- Arbitrary number of parameters (varargs)

```
def print_args(*items): # arguments are put in a tuple
    print(type(items))
    return items


>>> print_args(1,"hello",4.5)
<class 'tuple'>
(1, 'hello', 4.5)

#-----------------------------------

def print_kwargs(**items): # args are put in a dict
    print(type(items))
    return items


>>> print_kwargs(a=2,b=3,c=3)
<class 'dict'>
{'a': 2, 'b': 3, 'c': 3}
```

Functions are objects

- As everything in Python, also functions are object, of class **function**

```
def echo(arg): return arg

type(echo)          # <class 'function'>
hex(id(echo))       # 0x1003c2bf8
print(echo)         # <function echo at 0x1003c2bf8>
foo = echo
hex(id(foo))        # '0x1003c2bf8'
print(foo)          # <function echo at 0x1003c2bf8>
isinstance(echo, object)       # => True
```

Function documentation

- The comment after the functions header is bound to the **__doc__** special attribute

```
def my_function():
    """Summary line: do nothing, but document it.
    Description: No, really, it doesn't do anything.
    """
    pass


print(my_function.__doc__)
# Summary line: Do nothing, but document it.
#
#     Description: No, really, it doesn't do anything.

# try also 'help(my_function)'
```

Higher-order functions

- Functions can be passed as argument and returned as result
- Main combinators (**map**, **filter**) predefined: allow standard functional programming style in Python
- Heavy use of iterators, which support laziness
- Lambdas supported for use with combinators

```
lambda arguments: expression
```

  - The body can only be a single expression

Map

```
>>> print(map.__doc__)  % documentation
map(func, *iterables) --> map object
Make an iterator that computes the function using
arguments from each of the iterables.  Stops when the
shortest iterable is exhausted.
```

```
>>> map(lambda x:x+1, range(4))      % lazyness: returns
<map object at 0x10195b278>          % an iterator
>>> list(_)
[1, 2, 3, 4]
>>> list(map(lambda x, y : x+y, range(4), range(10)))
[0, 2, 4, 6]        % map of a binary function
>>> z = 5           % variable capture
>>> list(map(lambda x : x+z, range(4)))
[5, 6, 7, 8]
```

Map and List Comprehension

- **List comprehension** can replace uses of **map**

```
>>> list(map(lambda x:x+1, range(4)))
[1, 2, 3, 4]
>>> [x+1 for x in range(4)]
[1, 2, 3, 4]
>>> list(map(lambda x, y : x+y, range(4), range(10)))
[0, 2, 4, 6]    % map of a binary function
>>> [x+y for x in range(4) for y in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5,...  % NO!
>>> [x+y for (x,y) in zip(range(4),range(10))] % OK
[0, 2, 4, 6]
>>> print(zip.__doc__)
zip(iter1 [,iter2 [...]]) --> zip object
Return a zip object whose .__next__() method returns a tuple where
the i-th element comes from the i-th iterable argument. The
.__next__() method continues until the shortest iterable in the
argument sequence is exhausted and then it raises StopIteration.
```

Filter (and list comprehension)

```
>>> print(filter.__doc__)  % documentation
filter(function or None, iterable) --> filter object
Return an iterator yielding those items of iterable for
which function(item) is true. If function is None,
return the items that are true.
```

```
>>> filter(lambda x : x % 2 == 0,[1,2,3,4,5,6])
<filter object at 0x102288a58>  % lazyness
>>> list(_)                     % '_' is the last value
[2, 4, 6]
>>> [x for x in [1,2,3,4,5,6] if x % 2 == 0]
[2, 4, 6] % same using list comprehension
% How to say "false" in Python
>>> list(filter(None,
        [1,0,-1,"","Hello",None,[],[1],(),True,False]))
[1, -1, 'Hello', [1], True]
```

More modules for functional programming in Python

- **functools**: Higher-order functions and operations on callable objects, including:
  - **`reduce(`*`fun, iterable`*`[, `*`initializer`*`])`**
- **itertools**: Functions creating ***iterators*** for efficient looping. Inspired by constructs from APL, Haskell, and SML.
  - **`count(10) --> 10 11 12 13 14 ...`**
  - **`cycle('ABCD') --> A B C D A B C D ...`**
  - **`repeat(10, 3) --> 10 10 10`**
  - **`takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`**
  - **`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`**

Decorators

- A **decorator** is any callable Python object that is used to modify a **function**, **method** or **class definition**.
- A decorator is passed the original object being defined and returns a modified object, which is then bound to the name in the definition.
- (Function) Decorators exploit Python **higher-order features**:
  - Passing functions as argument
  - Nested definition of functions
  - Returning function
- Widely used in Python (system) programming
- Support several features of meta-programming

Basic idea: wrapping a function

```
def my_decorator(func):          # function as argument
    def wrapper(): # defines an inner function
        print("Something happens before the function.")
        func() # that calls the parameter
        print("Something happens after the function.")
    return wrapper # returns the inner function
```

```
def say_hello():   # a sample function
    print("Hello!")
# 'say_hello' is bound to the result of my_decorator
say_hello = my_decorator(say_hello) # function as arg
>>> say_hello()   # the wrapper is called
Something happens before the function.
Hello!
Something happens after the function.
```

Syntactic sugar: the "pie" syntax

```
def my_decorator(func):          # function as argument
    def wrapper(): # defines an inner function
        ... # as before
    return wrapper # returns the inner function
```

```
def say_hello():          ## HEAVY! 'say_hello' typed 3x
    print("Hello!")
say_hello = my_decorator(say_hello)
```

- Alternative, equivalent syntax

```
@my_decorator
def say_hello():
    print("Hello!")
```

Another decorator: do_twice

```
def do_twice(func):
    def wrapper_do_twice():
        func()       # the wrapper calls the
        func()       #      argument twice
    return wrapper_do_twice
```

```
@do_twice           # decorate the following
def say_hello():    # a sample function
    print("Hello!")
>>> say_hello()     # the wrapper is called
Hello!
Hello!
```

```
@do_twice          # does not work with parameters!!
def echo(str):     # a function with one parameter
    print(str)
>>> echo("Hi...") # the wrapper is called
TypErr: wrapper_do_twice() takes 0 pos args but 1 was given
>>> echo()
TypErr: echo() missing 1 required positional argument: 'str'
```

do_twice for functions with parameters

- Decorators for functions with parameters can be defined exploiting ***args** and ****kwargs**

```
def do_twice_args(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_do_twice
```

```
@do_twice_args
def say_hello():
    print("Hello!")
>>> say_hello()
Hello!
Hello!
```

```
@do_twice_args
def echo(str):
    print(str)
>>> echo("Hi... ")
Hi...
Hi...
```

General structure of a decorator

- Besides passing arguments, the wrapper also forwards the **result** of the decorated function
- Supports introspection redefining **__name__** and **__doc__**

```
import functools
def decorator(func):
    @functools.wraps(func)    #supports introspection
    def wrapper_decorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_decorator
```

Example: Measuring running time

```
import functools
import time

def timer(func):
    """Print the runtime of the decorated function"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()
        value = func(*args, **kwargs)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value
    return wrapper_timer

@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])
```

Other uses of decorators

- Debugging: prints argument list and result of calls to decorated function
- Registering plugins: adds a reference to the decorated function, without changing it
- In a web application, can wrap some code to check that the user is logged in
- @staticmethod and @classmethod make a function invocable on the class name or on an object of the class
- More: decorators can be nested, can have arguments, can be defined as classes…

Example: Caching Return Values

```
import functools
from decorators import count_calls

def cache(func):
    """Keep a cache of previous function calls"""
    @functools.wraps(func)
    def wrapper_cache(*args, **kwargs):
        cache_key = args + tuple(kwargs.items())
        if cache_key not in wrapper_cache.cache:
            wrapper_cache.cache[cache_key] = func(*args, **kwargs)
        return wrapper_cache.cache[cache_key]
    wrapper_cache.cache = dict()
    return wrapper_cache

@cache
@count_calls    # decorator that counts the invocations
def fibonacci(num):
    if num < 2:
        return num
    return fibonacci(num - 1) + fibonacci(num - 2)
```

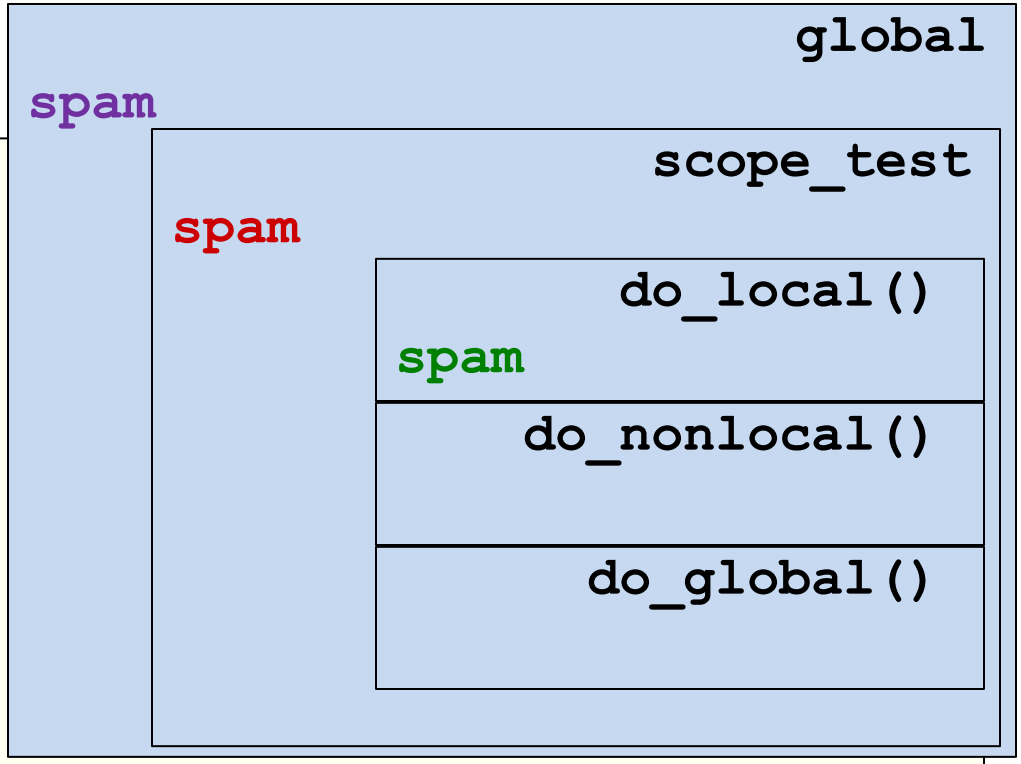
Namespaces and Scopes

- A ***namespace*** is a mapping from names to objects: typically implemented as a dictionary. Examples:
  - **builtins**: pre-defined functions, exception names,…
    - Created at intepreter's start-up
  - global names of a module
    - Created when the module definition is read
    - Note: names created in interpreter are in module __main__
  - local names of a function invocation
    - Created when function is called, deleted when it completes
  - and also names of a class, names of an object… see later
- Name ***x*** of a module ***m*** is an *attribute of* ***m***
  - accessible (read/write) with *"qualified name"* ***m.x***
  - if writable, it can be deleted with ***del***

Namespaces and Scopes (2)

- A **scope** is a textual region of a Python program where a namespace is **directly accessible**, i.e. reference to a name attempts to find the name in the namespace.
- Scopes are determined statically, but are used dynamically.
- During execution at least three namespaces are directly accessible, searched in the following order:
  - the scope containing the **local names**
  - the scopes of any enclosing functions, containing **non-local**, but also **non-global names**
  - the next-to-last scope containing the current module's **global names**
  - the outermost scope is the namespace containing **built-in names**
- **Assignments to names go in the local scope**
- Non-local variables can be accessed using nonlocal or global

Scoping rules




```
def scope_test():

    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"

    do_local()
    print("After local assignment:", spam)      # not affected
    do_nonlocal()
    print("After nonlocal assignment:", spam)   # affected
    do_global()
    print("After global assignment:", spam)     # not affected

scope_test()
print("In global scope:", spam)
```

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Criticisms to Python: scopes

- Control structures don't introduce a new scope

```
def test():
  for a in range(5):
    b = a % 2
    print(b)
  print(b)

>>> test()
```

```
def test(x):
  print(x)
  for x in range(5):
    print(x)
  print(x)

>>> test("Hello!")
```

Closures in Python

- Python supports closures: Even if the scope of the outer function is reclaimed on return, the non-local variables referred to by the nested function are saved in its attribute **`__closure__`**

```
def counter_factory():
  counter = 0
  def counter_increaser():
      nonlocal counter
      counter = counter + 1
      return counter
  return counter_increaser

>>> f = counter_factory()
>>> f()
1
>>> f()
2
>>> f.__closure__
(<cell at 0x1033ace88: int object at 0x10096dce0>,)
```