

Functions

- functions
- return
- scoping
- arguments
- keyword arguments
- *, **, ,
- global variables

(Simple) functions

- You can define your own functions using:

def *function-name* (*var*₁, ..., *var*_{*k*}):
 body code

*var*₁, ..., *var*_{*k*} are the *formal parameters*

- If the body code executes

return expression

the result of *expression* will be returned by the function. If expression is omitted or the body code terminates without performing `return`, then `None` is returned.

- When *calling* a function ***name*** (*value*, ..., *value*_{*k*}) body code is executed with *var*_{*i*}=*value*_{*i*}

Python shell

```
> def sum3(x, y, z):  
    return x+y+z  
  
> sum3(1, 2, 3)  
| 6  
> sum3(5, 7, 9)  
| 21  
  
> def powers(L, power):  
    P = [x**power for x in L]  
    return P  
  
> powers([2,3,4], 3)  
| [8, 27, 64]
```

Questions – `poly(3, "10", '3')` ?

```
def poly(z, x, y):  
    return z*x + y
```

- a) 33
- b) 1010103
- c) '33'
-  d) '1010103'
- e) TypeError
- f) Don't know

Why functions ?

Some other Python language features helping structuring programs

- Object orientation
- Modules
- Decorators
- Context managers
- Exceptions
- Doc strings
- doctest

- Avoid writting the same code multiple times, *re-usability*
- Be able to *name a functionality*
- Clearly state the functionality of a piece of code, *abstraction*:
Input = arguments, *output* = return value (and/or side effects)
- *Encapsulate* code with clear interface to the dependency to the outside world/code
- Share functionality in modules/libraries/packages with other users, *code sharing*
- Increase *readability* of code, smaller independent blocks of code
- Easier systematically *testing* of code
- ...

Local variables in functions

- The formal arguments and variables assigned to in the body of a function are created as temporary *local variables*

Global variables		Local variables	
sum3	<function>	x	4
a	3	y	5
y	42	z	6
		a	9
		b	15

state just before return b

Python shell

```
> def sum3(x, y, z):  
    a = x + y  
    b = a + z  
    return b  
  
> a = 3  
> y = 42  
> w = sum3(4, 5, 6)  
> w  
| 15  
> a  
| 3  
> b  
| NameError: name 'b' is not defined  
> x  
| NameError: name 'x' is not defined  
> y  
| 42  
> sum3  
| <function sum3 at 0x0356DA98>
```

Global variables

- Variables in function bodies that are only read, are considered access to *global variables*

Python shell

```
> prefix = "The value is"
> def nice_print(x):
    print(prefix, x)
> nice_print(7)
| The value is 7
> prefix = "Value ="
> nice_print(42)
| Value = 42
```

Global variables		Local variables	
nice_print	<function>	x	42
prefix	"Value ="		

state just before returning from 2nd nice_print

global

- Global variables that should be updated in the function body must be declared global in the body:

`global` *variable, variable, ...*

- Note: If you only need to read a global variable, it is not required to be declared global (but would be polite to the readers of your code)

Since `counter` assigned in body, `counter` will be considered to be a local variable

Python shell

```
> counter = 1

> def counted_print(x):
    global counter
    print("(%d)" % counter, x)
    counter += 1

> counted_print(7)
| (1) 7
> counted_print(42)
| (2) 42

> def counted_print(x):
    print("(%d)" % counter, x)
    counter += 1

> counted_print(7)
| UnboundLocalError: local variable
  'counter' referenced before
  assignment
```



Question – What value is printed ?

```
x = 1
def f(a):
    global x
    x = x + 1
    return a + x
print(f(2) + f(4))
```

a) 6

b) 7

c) 8

d) 9

e) 10



f) 11

g) 12

h) Don't know

Arbitrary number of arguments

- If you would like your function to be able to take a variable number of additional arguments in addition to the required, add a **variable* as the last argument.
- In a function call *variable* will be assigned a tuple with all the additional arguments.

Python shell

```
> def my_print(x, y, *L):  
    print("x =", x)  
    print("y =", y)  
    print("L =", L)  
  
> my_print(2, 3, 4, 5, 6, 7)  
| x = 2  
| y = 3  
| L = (4, 5, 6, 7)  
  
> my_print(42)  
| TypeError: my_print() missing 1  
| required positional argument: 'y'
```

Unpacking a list of arguments in a function call

- If you have list $\mathbb{L}$ (or tuple) containing the arguments to a *function call*, you can unpack them in the function call using $*\mathbb{L}$

$$\mathbb{L} = [x, y, z]$$
$$f(*\mathbb{L})$$

is equivalent to calling

$$f(\mathbb{L}[0], \mathbb{L}[1], \mathbb{L}[2])$$

i.e.

$$f(x, y, z)$$

- Note that $f(\mathbb{L})$ would pass a single argument to f , namely a list
- In a function call several $*$ expressions can appear, e.g. $f(*\mathbb{L}1, x, *\mathbb{L}2, *\mathbb{L}3)$

Python shell

```
> import math
> def norm(x, y):
    return math.sqrt(x * x + y * y)
> norm(3, 5)
| 5.830951894845301
> point = (3, 4)
> print(*point, sep=':')
| 3:4
> norm(point)
| TypeError: norm() missing 1 required positional argument: 'y'
> norm(*point)
| 5.0
> def dist(x0, y0, x1, y1):
    return math.sqrt((x1 - x0) ** 2 + (y1 - y0) ** 2)
> p = 3, 7
> q = 7, 4
> dist(p, q)
| TypeError: dist() missing 2 required positional arguments: 'x1' and 'y1'
> dist(*p, *q)
| 5.0
```

Question – How many arguments should `f` take ?

```
a = [1, 2, 3]
b = [4, 5]
c = (6, 7, 8)
d = (9, 10)
f(*a, b, c, *d)
```

a) 4

b) 5

c) 6



d) 7

e) 8

f) 9

g) 10

h) Don't know

Question – What is `list(zip(*zip(*L)))` ?

`L = [[1, 2, 3], [4, 5], [6, 7, 8]]`

a) `[([([1, 2, 3],),), ([4, 5],), ([6, 7, 8],),)]`

b) `[((1, 4, 6),), ((2, 5, 7),)]`



c) `[(1, 2), (4, 5), (6, 7)]`

d) `[(1, 2, 3), (4, 5), (6, 7, 8)]`

e) `[([1, 2, 3], [4, 5], [6, 7, 8]))]`

f) Don't know

Python shell

```
> list(zip((1, 2, 3), (4, 5, 6)))  
| [(1, 4), (2, 5), (3, 6)]
```

Keyword arguments

- Previously we have seen the following (strange) function calls

```
print(7, 14, 15, sep=":", end="")  
enumerate(my_list, start=1)
```

- name* = refers to one of the formal arguments, known as a **keyword argument**.
A *name* can appear at most once in a function call.
- In function calls, keyword arguments must follow positional arguments.
- Can e.g. be useful if there are many arguments, and the order is not obvious, i.e. improves readability of code:

```
complicated_function(  
    name = "Mickey",  
    city = "Duckburg",  
    state = "Calisota",  
    occupation = "Detective",  
    gender = "Male")
```

Python shell

```
> def sub(x, y):  
    return x - y  
  
> sub(9, 4)  
| 5  
  
> sub(y=9, x=4)  
| -5
```

Keyword arguments, default values

- When calling a function arguments can be omitted if the corresponding arguments in the function definition have default values *argument = value*.

Python shell

```
> def my_print(a, b, c=5, d=7):  
    print("a=%s, b=%s, c=%s, d=%s" % (a, b, c, d))  
  
> my_print(2, d=3, b=4)  
| a=2, b=4, c=5, d=3
```

Question – What is `f(6, z=2)` ?

```
def f(x, y=3, z=7):  
    return x + y + z
```

- a) 10
-  b) 11
- c) 16
- d) `TypeError: f() missing 1 required positional argument: 'y'`
- e) Don't know

Keyword arguments, mutable default values

- Be careful: Default value will be shared among calls (which can be useful)

The Python Language Reference 8.6 Function definitions

“Default parameter values are evaluated from left to right **when the function definition is executed**. This means that the expression is evaluated once, when the function is defined, and that the same “pre-computed” value is used for each call. This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default”

Python shell

```
> def list_append(e, L=[]):  
    L.append(e)  
    return L  
  
> list_append('x', ['y', 'z'])  
| ['y', 'z', 'x']  
> list_append("a")  
| ['a']  
> list_append("b")  
| ['a', 'b']  
> list_append("c")  
| ['a', 'b', 'c']
```



Python shell

```
> def list_append(e, L=None):  
    if L == None:  
        L = []  
    L.append(e)  
    return L  
  
> list_append('x', ['y', 'z'])  
| ['y', 'z', 'x']  
> list_append("a")  
| ['a']  
> list_append("b")  
| ['b']  
> list_append("c")  
| ['c']
```

Function call, dictionary of keyword arguments

- If you happen to have a *dictionary* containing the keyword arguments you want to pass to function, you can give all dictionary items as arguments using the single argument ***dictionary*

Python shell

```
> print(3, 4, 5, sep=":", end='#\n')  
| 3:4:5#  
> print_kwarg = {'sep': ':', 'end': '#\n'}  
> print(3, 4, 5, **print_kwarg)  
| 3:4:5#
```

Function definition, arbitrary keyword arguments

- If you want a function to accept arbitrary keyword arguments, add an argument `**argument` to the function definition.
- When the function is called *argument* will be assigned a dictionary containing the excess keyword arguments.

Python shell

```
> def my_print(a, b=3, **c):  
    print("a =", a)  
    print("b =", b)  
    print("c =", c)  
  
> my_print(x=27, y=42, a=7)  
| a = 7  
| b = 3  
| c = {'x': 27, 'y': 42}
```

Example

Python shell

```
> L1 = [1, 'a']
> L2 = ['b', 2, 3]
> D1 = {'y':4, 's':10}
> D2 = {'t':11, 'z':5.0}

> def f(a, b, c, d, e, *f, q=0, x=1, y=2, z=3, **kw):
    print("a=%s, b=%s, c=%s, d=%s, e=%s, " % (a, b, c, d, e),
          "f=%s\n" % str(f),
          "q=%s, x=%s, y=%s, z=%s, " % (q, x, y, z),
          "kw=%s" % kw,
          sep="")

> f(7, *L1, 9, *L2, x=7, **D1, w=42, **D2)
| a=7, b=1, c=a, d=9, e=b, f=(2, 3)
| q=0, x=7, y=4, z=5.0, kw={'w': 42, 's': 10, 't': 11}
```

non-keyword arguments must appear before keyword arguments

Forwarding function arguments

- `*` and `**` can e.g. be used to forward (unknown) arguments to other function calls

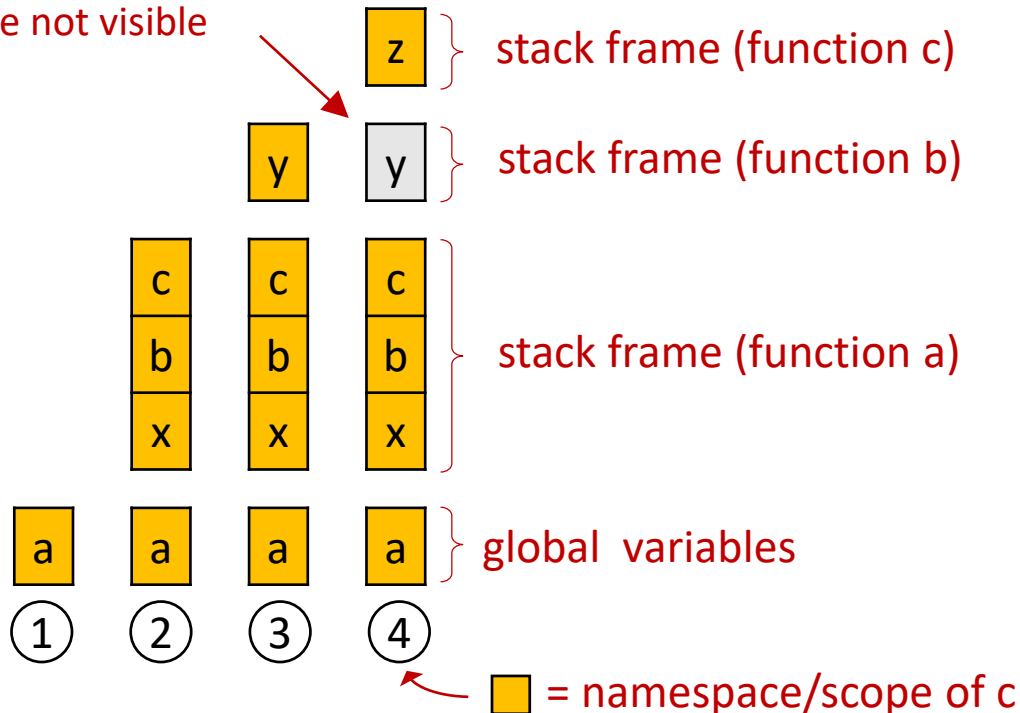
Python shell

```
> def my_print(*positional_arguments, sep=":", **keyword_arguments):  
    print(*positional_arguments, sep=sep, **keyword_arguments)  
  
> my_print(7, 42)  
| 7:42  
  
> my_print("x", "y", end="<")  
| x:y<  
  
> my_print("x", "y", sep="_")  
| x_y
```

Local function definitions and namespaces

- Function definitions can contain (nested) local function definitions, only accessible inside the function
- static/lexical scoping*, i.e. can see from the code which variables are in scope

in c, b's local variables
are not visible



Python shell

```
> def a(x):  
    def b(y):  
        print("b: y=%s x=%s" % (y, x))  
        c(y + 1)  
    def c(z):  
        print("c: z=%s x=%s" % (z, x))  
        print("a: x=%s" % x)  
        b(x + 1)  
  
> a(42)  
| a: x=42  
| b: y=43 x=42  
| c: z=44 x=42
```

Example – nested function definitions

Python shell

```
> def a(x):  
    def b(y):  
        print("Enter b (y=%s, x=%s)" % (y, x))  
        c(y + 1)  
        print("leaving b")  
    def c(x): # x hides argument of function a  
        def d(z):  
            print("Enter d (z=%s, x=%s)" % (z, x))  
            print("leaving d")  
        print("Enter c (x=%s)" % x)  
        d(x + 1)  
        print("leaving c")  
    print("Enter a (x=%s)" % x)  
    b(x + 1)  
    print("leaving a")
```

```
> a(5)  
| Enter a (x=5)  
| Enter b (y=6, x=5)  
| Enter c (x=7)  
| Enter d (z=8, x=7)  
| leaving d  
| leaving c  
| leaving b  
| leaving a
```

Example – nested functions and default values

Python shell

```
> def init_none(var_name):  
>     print('initializing', var_name)  
>     return None # redundant line  
  
> def f(a=init_none('a')):  
>     def g(b=init_none('b')):  
>         print('b =', b)  
>     print("a =", a)  
>     g(a + 1)  
  
| initializing a  
> f(10)  
| initializing b  
| a = 10  
| b = 11
```



nonlocal

- The `nonlocal` statement causes the listed identifiers to refer to previously bound variables in the nearest *enclosing scope excluding globals*.
- `nonlocal variable, variable, ...`

Python shell

```
> x = 0

> def f():
    y = 1
    def f_helper(z):
        global x
        nonlocal y
        print("(%s:%s) %s" % (x, y, z))
        y += 1
        x += 3

    f_helper(7)
    f_helper(42)

> f()
| (0:1) 7
| (3:2) 42
> f()
| (6:1) 7
| (9:2) 42
```

A note on Python and functions

- Similarities between Python and other languages:
 - functions are widely supported (sometimes called methods and procedures)
 - scoping rules is present in many languages (but details differ)
- The following are very Python specific (but nice):
 - how to handle global, local and nonlocal variables
 - keyword arguments
 - `*`, `**`