Python from Zero

Functions and Scope

User-Defined Functions

Subdivide your code into functions once it get's bigger

As soon as code repeats itself, consider writing a function

- Readability: Easier to follow code by encapsulating complex code in a simple function call
- Maintainability: If your code contains bugs, you really want this code to be in only one place to fix it!
- Portability: More easy to reuse function-code in other projects

User-Defined Functions

Basic:

```
def function_name( argument_list ):
    # indented code block
    return something
```

Different options to for the `argument_list`:

- Fixed number of mandatory arguments (may be empty)
- Fixed number of arguments, but some have default values (i.e. not mandatory)
- Variable number of unnamed arguments
- Variable number of keyword-arguments
- Variable number of unnamed and variable number of keyword-arguments

User-Defined Functions

Fixed number of mandatory arguments

```
def function_name():
    print("This function accepts no arguments at all")
    return

function_name()
```

```
def function_name(a, b, c):
    print("This function requires exactly three arguments")
    print(a,b,c)
    return

function_name(1,2,3)
function_name(a=1, b=2, c=3) # also possible to explicitly assign the arguments
function_name(b=2, a=1, c=3) # with this, argument order can be changed
```

User-Defined Functions

Fixed number of arguments, but some have default values

```
def function_name(a, b, c=None):
    print("This function requires two arguments and has one optional argument")
    if c is None:
        print(a,b)
    else:
        print(a,b,c)

    return

function_name(1,2)   # valid, c is None
function_name(1,2,3) # also valid, c is 3
```

This is illegal, *optional arguments must always come after the mandatory args*:

```
def function_name(a=1, b, c) # error
```

User-Defined Functions

Variable number of unnamed arguments: special syntax `*args`

```
def function_name(*args):
    print("args:")
    for arg in args:
        print(arg)

    return

function_name()
function_name(1,2,3)
function_name('a',1,'b',True)
```

```
function_name(a=1) # error, arguments have no name!
```

User-Defined Functions

Variable number of keyword-arguments: special syntax `**kwargs`

```
def function_name(**kwargs):
    print("kwargs:")
    for key in kwargs.keys():
        print(key+":", kwargs[key]) # kwargs is a dict!

    return

function_name()
function_name(a=1,b=2,c=3)
```

User-Defined Functions

Variable number of unnamed args and variable number of keyword-args

```
def function_name(*args, **kwargs):
    print("args:")
    for arg in args:
        print(arg)

    print("kwargs:")
    for key in kwargs.keys():
        print(key+":", kwargs[key])

    return

function_name()
function_name(1,2,c=3)
```

It's Your Turn!

- Read the "Addendum: User-defined Functions" in `Session_2.ipynb`

Do the exercise!

Exercise: Define a function named *compute* using a variable number of arguments (args) and a keyword argument named *operation* expecting one of the following strings: "add", "subtract", "multiply", "divide".

The function should then add/subtract/multiply/divide the numbers given by the preceding arguments. The computation should be performed pairwise e.g. first add *argument1* and *argument2*. Then add to the result *argument3*. Then add to the result *argument4* etc... Repeat for all arguments.

Hints:

- `*args`
- `if-elif-else`
- `for`, `range()`, `len()`

Scope

We have worked with names all the time in Python

```
x = 1 # x is a name

def my_function(): # my_function is a name
    # do stuff
    return
```

It is important to know which names can be used where: the *scope*

Scope

Four levels of name visibility:

- Built-in scope (don't worry too much about that)
  - Global scope
    - Enclosing scope
      - Local scope

The LEGB concept (Local, Enclosing, Global, Built-in)

Names defined in one scope are also visible in scopes below, but not above

- Python searches for names in LEGB order: first local, then enclosing, then global, then built-in
- If name cannot be found after that, we get an error

Scope

```
x = 1 # global scope

def func():
    print(x) # local scope, is enclosed by global, thus x is visible

def enclosing_func():
    y = 2 # enclosing scope (w.r.t. local_func)
    def local_func():
        z = x+y # local scope, can see x and y
        print(z)

    local_func()

print(x) # okay, same scope
print(y) # error, y is unknown here!
print(z) # error, z is unknown here!

func()           # okay, same scope
enclosing_func() # okay, same scope
local_func()     # error, unknown here
```

Scope

If you write a new script or start with a plain notebook, you are in *global scope*

If you write a function, the function body is a local scope

- The global scope is also the enclosing scope of the function
- Names defined in the function are not visible for the global scope

If you write a function inside a function...

- the inner function is a local scope
- the outer function is the enclosing scope of the inner function

Scope

Overwriting Names

In a local scope, names from the enclosing scope(s) can be re-used!

```
x = 1
def function():
    x = 'foo' # masks the global x and creates a new variable x in the local scope!
    print(x)  # accesses the local x

function()
print(x)      # accesses the global x as it does not know about the local scope
```

```
foo
1
```

Handle with care, can lead to confusions!

Scope

Pitfall: LEGB can be tricky

```
def my_function(my_variable):
    if my_Variable: # note the typo!
        # do important stuff
        print("done")
    else:
        print("nothing was done")

# do a lot of stuff and forget about my_variable

my_Variable = False
my_function(True)
```

What do you expect will happen?

Scope

Pitfall: LEGB can be tricky

```
def my_function(my_variable):
    if my_Variable: # note the typo! on execution, searches for `my_Variable` in LEGB order
        # do important stuff
        print("done")
    else:
        print("nothing was done")

# do a lot of stuff and forget about my_variable

my_Variable = False # global and defined by the time my_function is called
my_function(True)   # never uses the argument, but you also do not get an error!
```

```
nothing was done
```

Beware of this!

Scope

Loops and if-else clauses do not open up a new scope!

```
x = 0
for i in range(0,5):
    x = i    # does not mask x, but changes its value
    y = i**2 # creates new variable

print(x)
print(y) # y is visible

if (x<y):
    z = "foo"
else:
    a = "bar"

print(z) # z is visible
print(a) # else-branch was not executed, a was never created -> error!
```

It's Your Turn!

Go to our Moodle page (https://moodle.uni-greifswald.de/course/view.php?id=9565) and take the fifth quiz! ("Quiz 5 - Scope")