Leiden University
Medical Center

Python Programming

Functions

Outline

Introduction

User Defined Functions

Arguments

Variables Scope

Function as Values

Docstrings

Higher Order Functions

Anonymous Functions

Hands On!

Introduction

- A `function` is a named sequence of statements that performs some piece of work.
- Later on that function can be called multiple times by using its name.

User Defined Functions

Defining a function

A function definition includes its `name`, `parameters` (optional), and `body`:

```
def ␣name ( parameters ):
␣␣␣␣body
```

User Defined Functions

Defining a function

A function definition includes its `name`, `parameters` (optional), and `body`:

```
def ␣name ( parameters ):
␣␣␣␣body
```

functions.py

```python
def greeting():
    print('Hello!')
```

User Defined Functions

Calling a function

A function is called by using its `name` and by providing the required `arguments`:

```
name ( arguments )
```

User Defined Functions

Calling a function

A function is called by using its `name` and by providing the required `arguments`:

```
name ( arguments )
```

functions.py

```python
def greeting():
    print('Hello!')

greeting()
```

User Defined Functions

Calling a function

A function is called by using its `name` and by providing the required `arguments`:

```
name ( arguments )
```

functions.py

```python
def greeting():
    print('Hello!')

greeting()
```

terminal

```
$ python functions.py
Hello!
```

User Defined Functions

Calling a function

A function is called by using its `name` and by providing the required `arguments`:

```
name ( arguments )
```

Now let's add some `parameters`:

functions.py

```python
def greeting(name):
    print('Hello {}!'.format(name))

greeting('students')
```

terminal

```
$ python functions.py
Hello students !
```

User Defined Functions

The `return` statement

Used mainly to `return` a certain result value back to the caller.

functions.py

```python
def add_two(number):
    return number + 2
```

User Defined Functions

The `return` statement

Used mainly to `return` a certain result value back to the caller.

functions.py

```python
def add_two(number):
    return number + 2

print(add_two(5))
```

User Defined Functions

The `return` statement

Used mainly to `return` a certain result value back to the caller.

functions.py

```python
def add_two(number):
    return number + 2

print(add_two(5))
```

terminal

```
$ python functions.py
7
```

User Defined Functions

The `return` statement

Used mainly to `return` a certain result value back to the caller.

functions.py

```python
def add_two(number):
    return number + 2

for i in range(5):
    print('{} -> {}'.format(i, add_two(i)))
```

User Defined Functions

The `return` statement

Used mainly to `return` a certain result value back to the caller.

functions.py

```python
def add_two(number):
    return number + 2

for i in range(5):
    print('{} -> {}'.format(i, add_two(i)))
```

terminal

```
$ python functions.py
0 -> 2
1 -> 3
2 -> 4
3 -> 5
4 -> 6
```

User Defined Functions

The `return` statement

- Functions immediately `exit` when a `return` statement is encountered.
- No explicit value needs to be mentioned in the `return` statement.

User Defined Functions

The `return` statement

- Functions immediately `exit` when a `return` statement is encountered.
- No explicit value needs to be mentioned in the `return` statement.

negative.py

```python
def first_negative(numbers):
  for n in numbers:
     if n < 0:
       print(n)
       return
  print("No negative number found!")

first_negative([3, -5, 10, -2])
```

User Defined Functions

The `return` statement

- Functions immediately `exit` when a `return` statement is encountered.
- No explicit value needs to be mentioned in the `return` statement.

negative.py

```python
def first_negative(numbers):
  for n in numbers:
     if n < 0:
        print(n)
        return
  print("No negative number found!")

first_negative([3, -5, 10, -2])
```

terminal

```
$ python negative.py
-5
```

User Defined Functions

The `return` statement

- Something is always returned.

negative.py

```python
def first_negative(numbers):
  for n in numbers:
     if n < 0:
        print(n)
        return
  print("No negative number found!")

first_negative([3, -5, 10, -2])
```

terminal

```
$ python negative.py
-5
```

User Defined Functions

The `return` statement

- Something is always returned.

negative.py

```python
def first_negative(numbers):
  for n in numbers:
     if n < 0:
       print(n)
       return
  print("No negative number found!")

print(first_negative([3, -5, 10, -2]))
```

User Defined Functions

The `return` statement

- Something is always returned.

negative.py

```python
def first_negative(numbers):
  for n in numbers:
     if n < 0:
       print(n)
       return
  print("No negative number found!")

print(first_negative([3, -5, 10, -2]))
```

terminal

```
$ python negative.py
-5
None
```

User Defined Functions

The `return` statement

- Something is always returned, even if no `return` statement is reached.

negative.py

```python
def first_negative(numbers):
  for n in numbers:
     if n < 0:
       print(n)
       return
  print("No negative number found!")

print(first_negative([]))
```

User Defined Functions

The `return` statement

- Something is always returned, even if no `return` statement is reached.

negative.py

```python
def first_negative(numbers):
  for n in numbers:
     if n < 0:
       print(n)
       return
  print("No negative number found!")

print(first_negative([]))
```

terminal

```
$ python negative.py
No negative number found!
None
```

Arguments

Required

Have to be passed during the function call (precisely in the right order).

functions.py

```python
def add_two(number):
    return number + 2

print(add_two())
```

Arguments

Required

Have to be passed during the function call (precisely in the right order).

functions.py

```python
def add_two(number):
    return number + 2

print(add_two())
```

terminal

```
$ python functions.py
File "functions.py", line 4, in <module>
add_two()
TypeError:  add_two() missing 1 required positional argument:  ’number’
```

Arguments

Default

Have a default value if no argument value is passed during the function call.

functions.py

```python
def add_value(number, default=2):
    return number + default

print(add_value(5))
```

terminal

```
$ python functions.py
7
```

Arguments

Default

Have a default value if no argument value is passed during the function call.

functions.py

```python
def add_value(number, default=2):
    return number + default

print(add_value(5))
print(add_value(5, 5))
```

terminal

```
$ python functions.py
7
10
```

Arguments

Explicit parameter mentioning

When you want to make sure that the mapping is correct.

functions.py

```python
def add_value(number, default=2):
    return number + default

print(add_value(5, default=2))
print(add_value(number=5, default=2))
print(add_value(default=2, number=5))
```

terminal

```
$ python functions.py
7
7
7
```

Variables Scope

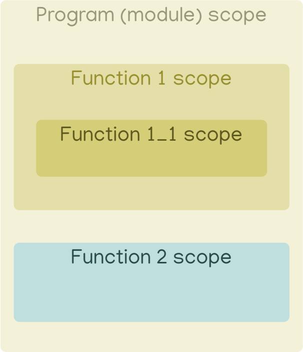
Scope refers to the variables visibility, i.e., in which program parts can be seen and used.

Variables Scope

Scope refers to the variables visibility, i.e., in which program parts can be seen and used.

Roughly speaking:

- The whole program (module) forms one scope.
- A function definition creates a new (nested) scope.

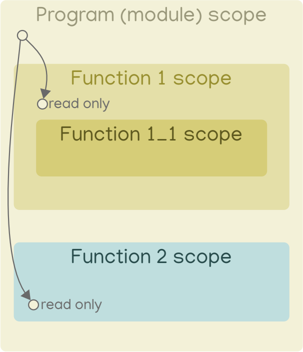


Variables Scope

Scope refers to the variables visibility, i.e., in which program parts can be seen and used.

Roughly speaking:

- The whole program (module) forms one scope.
- A function definition creates a new (nested) scope.
- Variables from an outside scope are visible in the inner nested scope, but you cannot (re)-assign a value to them (read only) unless they are declared global.

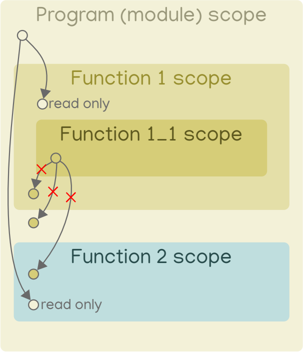


Variables Scope

Scope refers to the variables visibility, i.e., in which program parts can be seen and used.

Roughly speaking:

- The whole program (module) forms one scope.
- A function definition creates a new (nested) scope.
- Variables from an outside scope are visible in the inner nested scope, but you cannot (re)-assign a value to them (read only) unless they are declared global.
- Variables inside a nested scope are not visible in the outer scope.



Variables Scope

scope.py

```python
g1 = 0
if g1 == 0:
    g2 = 1

def some_function(p):
    l = 3
    print(p)
    print(l)

# Calling the function
some_function(23)

print(p, l)

print(g1, g2)
```

Variables Scope

scope.py

Module scope

```python
g1 = 0
if g1 == 0:
    g2 = 1

def some_function(p):
    l = 3
    print(p)
    print(l)

# Calling the function
some_function(23)

print(p, l)

print(g1, g2)
```

Variables Scope

scope.py

Module scope

```python
g1 = 0          # A global variable
if g1 == 0:
    g2 = 1

def some_function(p):
    l = 3
    print(p)
    print(l)

# Calling the function
some_function(23)

print(p, l)

print(g1, g2)
```

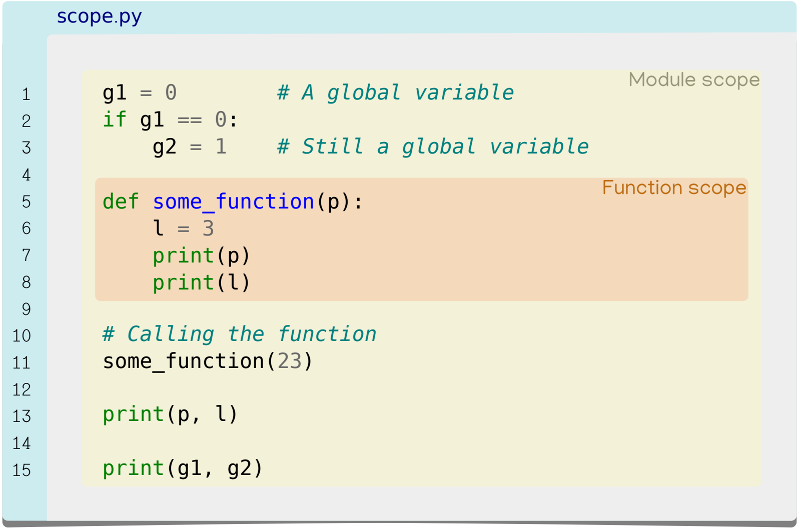
Variables Scope

scope.py

Module scope

```python
g1 = 0          # A global variable
if g1 == 0:
    g2 = 1      # Still a global variable

def some_function(p):
    l = 3
    print(p)
    print(l)

# Calling the function
some_function(23)

print(p, l)

print(g1, g2)
```

Variables Scope



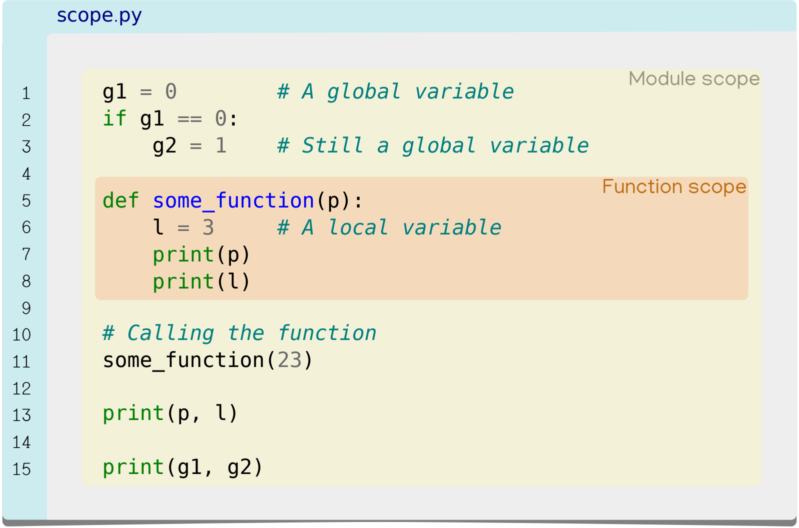
scope.py

Module scope

Function scope

```
g1 = 0          # A global variable
if g1 == 0:
    g2 = 1      # Still a global variable

def some_function(p):
    l = 3
    print(p)
    print(l)

# Calling the function
some_function(23)

print(p, l)

print(g1, g2)
```

Variables Scope



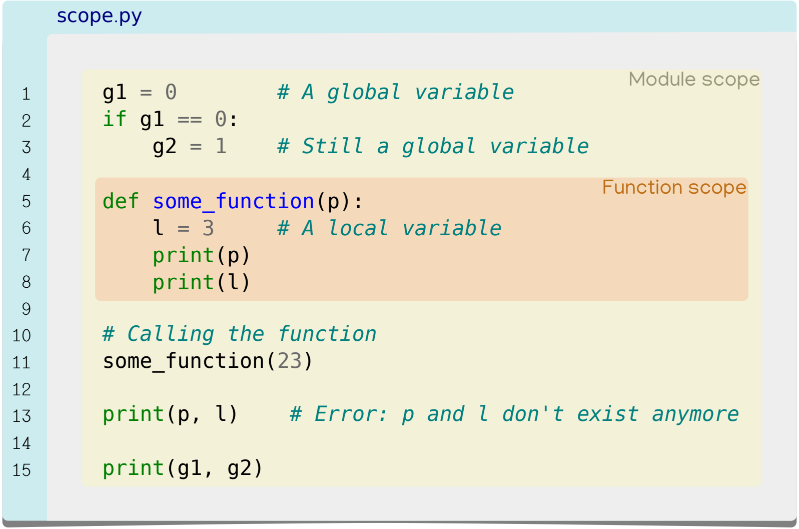
scope.py

Module scope

```python
g1 = 0          # A global variable
if g1 == 0:
    g2 = 1      # Still a global variable

def some_function(p):
    l = 3       # A local variable
    print(p)
    print(l)

# Calling the function
some_function(23)

print(p, l)

print(g1, g2)
```

Function scope

Variables Scope



scope.py

Module scope

Function scope

```python
g1 = 0        # A global variable
if g1 == 0:
    g2 = 1    # Still a global variable

def some_function(p):
    l = 3     # A local variable
    print(p)
    print(l)

# Calling the function
some_function(23)

print(p, l)    # Error: p and l don't exist anymore

print(g1, g2)
```

Variables Scope

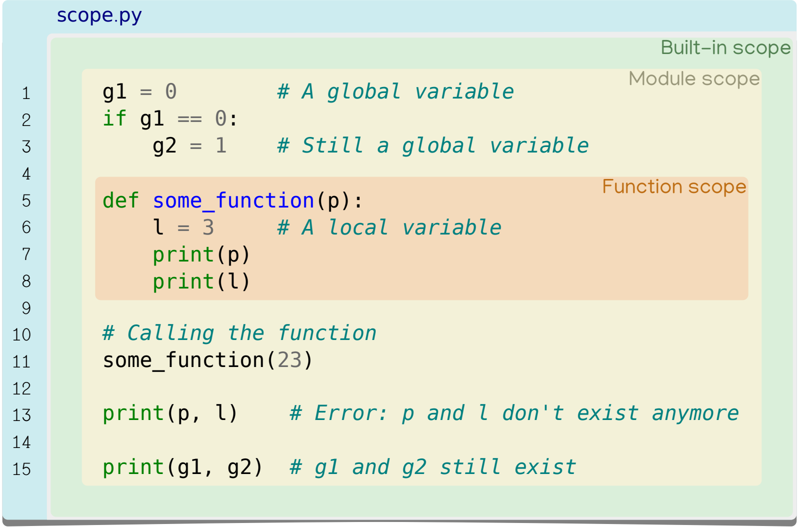
scope.py

Module scope

```python
g1 = 0          # A global variable
if g1 == 0:
    g2 = 1      # Still a global variable

def some_function(p):
    l = 3       # A local variable
    print(p)
    print(l)

# Calling the function
some_function(23)

print(p, l)      # Error: p and l don't exist anymore

print(g1, g2)  # g1 and g2 still exist
```

Function scope

Variables Scope



scope.py

```python
g1 = 0          # A global variable
if g1 == 0:
    g2 = 1      # Still a global variable

def some_function(p):
    l = 3       # A local variable
    print(p)
    print(l)

# Calling the function
some_function(23)

print(p, l)     # Error: p and l don't exist anymore

print(g1, g2)  # g1 and g2 still exist
```

Variables Scope

Hiding variables

If in a new scope a variable is created that already exists in an outer scope, the new variable will hide the outer variable.

Variables Scope

Hiding variables

If in a new scope a variable is created that already exists in an outer scope, the new variable will hide the outer variable.

scope_hiding.py

```python
a = 1

def some_function():
    a = 2 # Hides the global a variable
    print(a)

# Calling the function
some_function()
print(a)
```

Variables Scope

Hiding variables

If in a new scope a variable is created that already exists in an outer scope, the new variable will hide the outer variable.

scope_hiding.py

```python
a = 1

def some_function():
    a = 2 # Hides the global a variable
    print(a)

# Calling the function
some_function()
print(a)
```

terminal

```
$ python scope_hiding.py
2
1
```

Variables Scope

Hiding variables

If in a new scope a variable is created that already exists in an outer scope, the new variable will hide the outer variable.

scope_hiding.py

```python
a = 1

def some_function():
    a = 2 # Hides the global a variable
    print(a)

# Calling the function
some_function()
print(a)
```

terminal

```
$ python scope_hiding.py
2
1
```

This applies to function parameters as well.

Variables Scope

The `global` keyword

Allows a variable to be changed outside of the current scope.

Variables Scope

The `global` keyword

Allows a variable to be changed outside of the current scope.

global.py

```python
a = 1

def some_function():
    global a  # a is the global one
    a = 2
    print(a)

# Calling the function
some_function()
print(a)
```

Variables Scope

The `global` keyword

Allows a variable to be changed outside of the current scope.

global.py

```python
a = 1

def some_function():
    global a  # a is the global one
    a = 2
    print(a)

# Calling the function
some_function()
print(a)
```

terminal

```
$ python global.py
2
2
```

Variables Scope

What about parameters and arguments?

parameters.py

```python
def some_function(b):
    b = 2
    print(b)

a = 1

print(a)

# Calling the function
some_function(a)

print(a)
```

Variables Scope

What about parameters and arguments?

parameters.py

```python
def some_function(b):
    b = 2
    print(b)

a = 1

print(a)

# Calling the function
some_function(a)

print(a)
```

terminal

```
$ python parameters.py
1
2
1
```

Variables Scope

Mutable arguments

mutable_params.py

```python
def some_function(a_list):
    a_list = 13
    print('a_list:', a_list)

a = [7, 5]

print('a before the call:', a)

# Calling the function
some_function(a)

print('a after the call:', a)
```

Variables Scope

Mutable arguments

mutable_params.py

```python
def some_function(a_list):
    a_list = 13
    print('a_list:', a_list)

a = [7, 5]

print('a before the call:', a)

# Calling the function
some_function(a)

print('a after the call:', a)
```

terminal

```
$ python mutable_params.py
a before the call:  [7, 5]
a_list:  13
a after the call:  [7, 5]
```

Variables Scope

Mutable arguments

However, element changes to update the argument.

mutable_params.py

```python
def some_function(a_list):
    a_list[1] = 13
    print('a_list:', a_list)

a = [7, 5]

print('a before the call:', a)

# Calling the function
some_function(a)

print('a after the call:', a)
```

Variables Scope

Mutable arguments

However, element changes to update the argument.

mutable_params.py

```python
def some_function(a_list):
    a_list[1] = 13
    print('a_list:', a_list)

a = [7, 5]

print('a before the call:', a)

# Calling the function
some_function(a)

print('a after the call:', a)
```

terminal

```
$ python mutable_params.py
a before the call:  [7, 5]
a_list:  [7, 13]
a after the call:  [7, 13]
```

Function as Values

We can pass functions around just like other values, and call them.

function_values.py

```python
def add_two(number):
    return number + 2

def add_some_other_number(number, other_number=12):
    return number + other_number

functions = [add_two, add_some_other_number]
for function in functions:
    print(function(7))
```

Function as Values

We can pass functions around just like other values, and call them.

function_values.py

```python
def add_two(number):
    return number + 2

def add_some_other_number(number, other_number=12):
    return number + other_number

functions = [add_two, add_some_other_number]
for function in functions:
    print(function(7))
```

terminal

```
$ python function_values.py
9
19
```

Docstrings

- Regular string values which you start the function definition body with.
- You can access a docstring using the `help()` built-in.

docstrings.py

```python
def factorial(n):
    """Compute factorial of n in the obvious way."""
    if n == 0:
        return 1
    else:
        return factorial(n - 1) * n

help(factorial)
```

Docstrings

- Regular string values which you start the function definition body with.
- You can access a docstring using the `help()` built-in.

docstrings.py

```python
def factorial(n):
    """Compute factorial of n in the obvious way."""
    if n == 0:
        return 1
    else:
        return factorial(n - 1) * n

help(factorial)
```

terminal

```
$ python docstrings.py
Help on function factorial in module __main__:
factorial(n)
Compute factorial of n in the obvious way.
```

Higher Order Functions

Take a function as an argument.

IPython

```
In [1]:  help(map)
         Help on class map in module builtins:
         class map(object)
           |  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.
In [2]:  list(map(add_two, [1, 2, 3, 4]))
Out[2]:  [3, 4, 5, 6]
```

Anonymous Functions

Can be created with the `lambda` keyword:

```
lambda ␣parameters : ␣expression
```

anonymous.py

```python
x_times_7 = lambda x: x * 7
print(x_times_7(4))
```

Anonymous Functions

Can be created with the `lambda` keyword:

```
lambda ␣parameters : ␣expression
```

anonymous.py

```python
x_times_7 = lambda x: x * 7
print(x_times_7(4))
```

terminal

```
$ python anonymous.py
28
```

Anonymous Functions

Can be created with the `lambda` keyword:

```
lambda ␣parameters : ␣expression
```

anonymous.py

```python
x_times_7 = lambda x: x * 7
print(x_times_7(4))

for i in list(map(lambda x: x+2, range(4))):
    print(i)
```

Anonymous Functions

Can be created with the `lambda` keyword:

```
lambda ␣parameters : ␣expression
```

anonymous.py

```python
x_times_7 = lambda x: x * 7
print(x_times_7(4))

for i in list(map(lambda x: x+2, range(4))):
    print(i)
```

terminal

```
$ python anonymous.py
28
2
3
4
5
```

Wrap up

You are now able to:

- Create yout own user defined functions.
- Call your defined functions with the right arguments.
- Understand variables scope.
- Write docstrings for your functions.
- Use higher order functions.
- Employ anonymous functions.

Hands On!

1. Write a Python function that returns the maximum of two numbers.
2. Write a Python function that returns the maximum of three numbers. Try to reuse the first maximum of two numbers function.
3. Write a Python function that accepts a list as a parameter. Next, it determines and prints the number of positive and negative numbers.

Acknowledgements

Martijn Vermaat
Jeroen Laros
Jonathan Vis