Python Programming

Functions

Outline

- 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.

Defining a function

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

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

Defining a function

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

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

functions.py

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

Calling a function

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

```
name ( arguments )
```

Calling a function

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

```
name ( arguments )
```

```
functions.py
1  def greeting():
2      print('Hello!')
3
4  greeting()
```

Calling a function

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

```
name ( arguments )
```

functions.py

```python
1  def greeting():
2      print('Hello!')
3
4  greeting()
```

terminal

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

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
1  def greeting(name):
2      print('Hello {}!'.format(name))
3
4  greeting('students')
```

terminal

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

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
```

The `return` statement

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

```python
functions.py
1  def add_two(number):
2      return number + 2
3
4  print(add_two(5))
```

The `return` **statement**

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

functions.py

```python
1  def add_two(number):
2      return number + 2
3
4  print(add_two(5))
```

terminal

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

The `return` statement

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

```python
functions.py
1  def add_two(number):
2      return number + 2
3
4  for i in range(5):
5      print('{} -> {}'.format(i, add_two(i)))
```

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
```

The `return` statement

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

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])
```

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
```

The `return` **statement**

- Something is always returned.

negative.py

```python
1  def first_negative(numbers):
2    for n in numbers:
3        if n < 0:
4          print(n)
5          return
6    print("No negative number found!")
7
8  first_negative([3, -5, 10, -2])
```

terminal

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

The `return` **statement**

- Something is always returned.

negative.py

```python
1  def first_negative(numbers):
2    for n in numbers:
3        if n < 0:
4          print(n)
5          return
6    print("No negative number found!")
7
8  print(first_negative([3, -5, 10, -2]))
```

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
```

The `return` **statement**

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

negative.py
```python
1  def first_negative(numbers):
2    for n in numbers:
3       if n < 0:
4         print(n)
5         return
6    print("No negative number found!")
7
8  print(first_negative([]))
```

The `return` **statement**

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

```
negative.py
1  def first_negative(numbers):
2    for n in numbers:
3      if n < 0:
4        print(n)
5        return
6    print("No negative number found!")
7
8  print(first_negative([]))
```

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

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())
```

Required

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

functions.py

```
1  def add_two(number):
2      return number + 2
3
4  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'
```

Default

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

functions.py

```python
1  def add_value(number, default=2):
2      return number + default
3
4  print(add_value(5))
```

terminal

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

Default

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

functions.py

```python
1  def add_value(number, default=2):
2      return number + default
3
4  print(add_value(5))
5  print(add_value(5, 5))
```

terminal

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

Explicit parameter mentioning

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

```python
functions.py

1  def add_value(number, default=2):
2      return number + default
3
4  print(add_value(5, default=2))
5  print(add_value(number=5, default=2))
6  print(add_value(default=2, number=5))
```
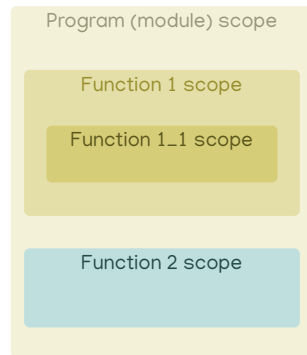
```
terminal

$ python functions.py
7
7
7
```

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

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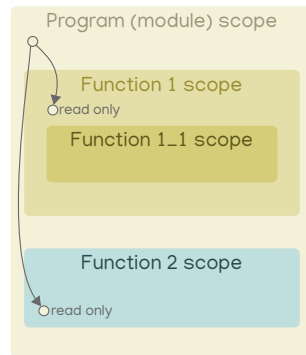
Roughly speaking:

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

Program (module) scope

Function 1 scope

Function 1_1 scope

Function 2 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.

Program (module) scope

Function 1 scope
read only

Function 1_1 scope
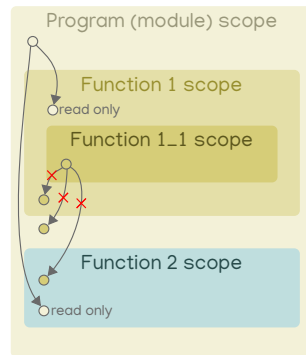
Function 2 scope

read only

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.

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)
```

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)
```

Module scope

Variables Scope

scope.py

```python
g1 = 0          # A global variable          Module scope
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

```
1    g1 = 0          # A global variable          Module scope
2    if g1 == 0:
3        g2 = 1      # Still a global variable
4
5    def some_function(p):
6        l = 3
7        print(p)
8        print(l)
9
10   # Calling the function
11   some_function(23)
12
13   print(p, l)
14
15   print(g1, g2)
```

Variables Scope

scope.py

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

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

# Calling the function
some_function(23)

print(p, l)

print(g1, g2)
```

scope.py

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

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

# Calling the function
some_function(23)

print(p, l)

print(g1, g2)
```

scope.py

```python
1    g1 = 0          # A global variable      Module scope
2    if g1 == 0:
3        g2 = 1      # Still a global variable
4
5    def some_function(p):                    Function scope
6        l = 3       # A local variable
7        print(p)
8        print(l)
9
10   # Calling the function
11   some_function(23)
12
13   print(p, l)     # Error: p and l don't exist anymore
14
15   print(g1, g2)
```

Variables Scope

scope.py

```python
1    g1 = 0          # A global variable        Module scope
2    if g1 == 0:
3        g2 = 1      # Still a global variable
4
5    def some_function(p):                      Function scope
6        l = 3       # A local variable
7        print(p)
8        print(l)
9
10   # Calling the function
11   some_function(23)
12
13   print(p, l)     # Error: p and l don't exist anymore
14
15   print(g1, g2)   # g1 and g2 still exist
```

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
```

Built–in scope

Module scope

Function 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.

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
1   a = 1
2
3   def some_function():
4       a = 2 # Hides the global a variable
5       print(a)
6
7   # Calling the function
8   some_function()
9   print(a)
```

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

```
1  a = 1
2
3  def some_function():
4      a = 2 # Hides the global a variable
5      print(a)
6
7  # Calling the function
8  some_function()
9  print(a)
```

terminal

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

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.

```python
scope_hiding.py
1   a = 1
2
3   def some_function():
4       a = 2 # Hides the global a variable
5       print(a)
6
7   # Calling the function
8   some_function()
9   print(a)
```

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

This applies to function parameters as well.

The `global` **keyword**

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

The `global` keyword

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

```python
global.py
1   a = 1
2
3   def some_function():
4       global a   # a is the global one
5       a = 2
6       print(a)
7
8   # Calling the function
9   some_function()
10  print(a)
```

The `global` keyword

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

```python
a = 1

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

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

global.py

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

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)
```

What about parameters and arguments?

parameters.py

```python
1  def some_function(b):
2      b = 2
3      print(b)
4
5  a = 1
6
7  print(a)
8
9  # Calling the function
10 some_function(a)
11
12 print(a)
```

terminal

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

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)
```

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]
```

Mutable arguments

However, element changes to update the argument.

```python
mutable_params.py
1  def some_function(a_list):
2      a_list[1] = 13
3      print('a_list:', a_list)
4
5  a = [7, 5]
6
7  print('a before the call:', a)
8
9  # Calling the function
10 some_function(a)
11
12 print('a after the call:', a)
```

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]
```

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

function_values.py

```python
1  def add_two(number):
2      return number + 2
3
4  def add_some_other_number(number, other_number=12):
5      return number + other_number
6
7  functions = [add_two, add_some_other_number]
8  for function in functions:
9      print(function(7))
```

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

function_values.py

```
1  def add_two(number):
2      return number + 2
3
4  def add_some_other_number(number, other_number=12):
5      return number + other_number
6
7  functions = [add_two, add_some_other_number]
8  for function in functions:
9      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
1  def factorial(n):
2      """Compute factorial of n in the obvious way."""
3      if n == 0:
4          return 1
5      else:
6          return factorial(n - 1) * n
7
8  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
1  def factorial(n):
2      """Compute factorial of n in the obvious way."""
3      if n == 0:
4          return 1
5      else:
6          return factorial(n - 1) * n
7
8  help(factorial)
```

terminal

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

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]
```

Can be created with the `lambda` keyword:

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

anonymous.py

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

Can be created with the `lambda` keyword:

```
lambda parameters : expression
```

anonymous.py

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

terminal

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

Can be created with the `lambda` keyword:

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

anonymous.py

```python
1   x_times_7 = lambda x: x * 7
2   print(x_times_7(4))
3
4   for i in list(map(lambda x: x+2, range(4))):
5       print(i)
```

Can be created with the `lambda` keyword:

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

anonymous.py

```python
1  x_times_7 = lambda x: x * 7
2  print(x_times_7(4))
3
4  for i in list(map(lambda x: x+2, range(4))):
5      print(i)
```

terminal

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

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