Leiden University
Medical Center

Python Programming

Functions

Mihai Lefter

Outline

User Defined Functions

- 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

```
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

```
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

```
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

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

The `return` statement

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

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

```
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.

functions.py

```
1  def add_two(number):
2      return number + 2
3
4  for i in range(5):
5      print(i, '->', add_two(i))
```

The `return` statement

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

functions.py

```
1  def add_two(number):
2      return number + 2
3
4  for i in range(5):
5      print(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.

functions.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  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.

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

terminal

```
$ python functions.py
-5
```

The `return` statement

- Something is always returned.

functions.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([3, -5, 10, -2]))
```

The `return` statement

- Something is always returned.

functions.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([3, -5, 10, -2]))
```

terminal

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

The `return` statement

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

functions.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([]))
```

The `return` statement

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

functions.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 functions.py
No negative number found!
None
```

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())
```

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

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

functions.py

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

terminal

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

Default

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

functions.py

```
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.

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
```

Variable Scope

The range where a variable is visible.

Roughly speaking:

- The whole program forms one scope.
- A function definition creates a new (nested) scope.
- Variables inside a nested scope are not visible in the outer scope.
- Variables from the 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.

Hiding variables

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

Scope of function arguments

Arguments of a function declaration are also only visible inside the function.

Functions are values

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))
10
11 # Simple anonymous functions can be created with lambda.
12 functions.append(lambda x: x * 7)
13 for function in functions:
14     print(function(4))
```

Docstrings

Like many other definitions, functions can have docstrings.

- Docstrings are regular string values which you start the definition body with.
- You can access an object's docstring using help.

docstring_example.py

```
1  def factorial(n):
2      """Compute factorial of n in the obious way."""
3      if n == 0:
4          return 1
5      else:
6          return factorial(n - 1) * n
```

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]
```

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 string as parameter. Next, it calculates and prints the number of upper case letters and lower case letters. Make use of the `isupper` and `islower` built in methods.

Leiden University
Medical Center

Acknowledgements

Martijn Vermaat
Jeroen Laros
Jonathan Vis