Introduction to Python - II

Overview

- Error management
- Built-in functions
- Modules and packages
- Namespaces
- Standard modules and packages
- Real world example

Introduction to Python - II

Overview

- Error management
- Built-in functions
- Modules and packages
- Namespaces
- Standard modules and packages
- Real world example

Exceptions -1

Errors in Python programs are managed by means of exceptions.

Exceptions -1

Errors in Python programs are managed by means of exceptions.

file: `error.py`

```
def division(a, b):
    return a/b
```

Exceptions -1

Errors in Python programs are managed by means of exceptions.

file: `error.py`

```
def division(a, b):
    return a/b
```

Now let's force an error:

```
>>> from error import division
>>> division(2.0, 4)
0.5
>>> division(2.0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error.py", line 2, in division
    return a/b
ZeroDivisionError: float division by zero
```

Exceptions -1

Errors in Python programs are managed by means of exceptions.

file: `error.py`

```
def division(a, b):
    return a/b
```

Now let's force an error:

```
>>> from error import division
>>> division(2.0, 4)
0.5
>>> division(2.0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error.py", line 2, in division
    return a/b
ZeroDivisionError: float division by zero
```

1 ← division(2.0, 0)
2 ← Traceback (most recent call last):
2.1 ← File "<stdin>", line 1, in <module>
2.2 ← File "error.py", line 2, in division
2.3 ← ZeroDivisionError: float division by zero

Exceptions -1

Errors in Python programs are managed by means of exceptions.

file: `error.py`

```
def division(a, b):
    return a/b
```

Now let's force an error:

```
>>> from error import division
>>> division(2.0, 4)
0.5
>>> division(2.0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error.py", line 2, in division
    return a/b
ZeroDivisionError: float division by zero
```

1 ← 1
2 ← 2
2.1 ← 2.1
2.2 ← 2.2
2.3 ← 2.3

1. When an exception occurs, by default it is managed by the Python interpreter

Exceptions -1

Errors in Python programs are managed by means of exceptions.

file: `error.py`

```
def division(a, b):
    return a/b
```

Now let's force an error:

```
>>> from error import division
>>> division(2.0, 4)
0.5
>>> division(2.0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error.py", line 2, in division
    return a/b
ZeroDivisionError: float division by zero
```

1 ← division(2.0, 0)
2 ← Traceback (most recent call last):
2.1 ← File "<stdin>", line 1, in <module>
2.2 ← File "error.py", line 2, in division
2.3 ← ZeroDivisionError: float division by zero

1. When an exception occurs, by default it is managed by the Python interpreter
2. The execution is stopped, and some information is provided:

Exceptions -1

Errors in Python programs are managed by means of exceptions.

file: `error.py`

```
def division(a, b):
    return a/b
```

Now let's force an error:

```
>>> from error import division
>>> division(2.0, 4)
0.5
>>> division(2.0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error.py", line 2, in division
    return a/b
ZeroDivisionError: float division by zero
```

1 ← 1
2 ← 2
2.1 ← 2.1
2.2 ← 2.2
2.3 ← 2.3

1. When an exception occurs, by default it is managed by the Python interpreter
2. The execution is stopped, and some information is provided:
   1. The point in the program where error occurred

Exceptions -1

Errors in Python programs are managed by means of exceptions.

file: `error.py`

```
def division(a, b):
    return a/b
```

Now let's force an error:

```
>>> from error import division
>>> division(2.0, 4)
0.5
>>> division(2.0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error.py", line 2, in division
    return a/b
ZeroDivisionError: float division by zero
```

1 ← 1
2 ← 2
2.1 ← 2.1
2.2 ← 2.2
2.3 ← 2.3

1. When an exception occurs, by default it is managed by the Python interpreter
2. The execution is stopped, and some information is provided:
   1. The point in the program where error occurred
   2. The statement which caused the error

Exceptions -1

Errors in Python programs are managed by means of exceptions.

file: `error.py`

```
def division(a, b):
    return a/b
```

Now let's force an error:

```
>>> from error import division
>>> division(2.0, 4)
0.5
>>> division(2.0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error.py", line 2, in division
    return a/b
ZeroDivisionError: float division by zero
```

1 ← 1
2 ← 2
2.1 ← 2.1
2.2 ← 2.2
2.3 ← 2.3

1. When an exception occurs, by default it is managed by the Python interpreter
2. The execution is stopped, and some information is provided:
   1. The point in the program where error occurred
   2. The statement which caused the error
   3. And the type of error

Exceptions -1

Errors in Python programs are managed by means of exceptions.

file: `error.py`

```
def division(a, b):
    return a/b
```

Now let's force an error:

```
>>> from error import division
>>> division(2.0, 4)
0.5
>>> division(2.0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error.py", line 2, in division
    return a/b
ZeroDivisionError: float division by zero
```


1. When an exception occurs, by default it is managed by the Python interpreter
2. The execution is stopped, and some information is provided:
   1. The point in the program where error occurred
   2. The statement which caused the error
   3. And the type of error

→

Exceptions -2

Exceptions can be managed by ”catching” them:

Exceptions -2

Exceptions can be managed by ”catching” them:

file: `error1.py`

```
import error

def division(a,b):
    try:
        return error.division(a, b)
    except:
        print("You can't divide by zero!")
```

Exceptions -2

Exceptions can be managed by ”catching” them:

file: `error1.py`

```
import error   <-- 1

def division(a,b):
    try:
        return error.division(a, b)   <-- 2
    except:
        print("You can't divide by zero!")   <-- 3
```

Exceptions -2

Exceptions can be managed by ”catching” them:

file: `error1.py`

```
import error   <-- 1

def division(a,b):
    try:
        return error.division(a, b)   <-- 2
    except:
        print("You can't divide by zero!")   <-- 3
```

1. Note: a new way to import functions

Exceptions -2

Exceptions can be managed by ”catching” them:

file: `error1.py`

```
import error   <-- 1

def division(a,b):
    try:
        return error.division(a, b)   <-- 2
    except:
        print("You can't divide by zero!")   <-- 3
```

1. Note: a new way to import functions
2. If an exception happens here ...

Exceptions -2

Exceptions can be managed by ”catching” them:

file: `error1.py`

```
import error   <-- 1

def division(a,b):
    try:
        return error.division(a, b)   <-- 2
    except:
        print("You can't divide by zero!")   <-- 3
```

1. Note: a new way to import functions
2. If an exception happens here ...
3. .. do this

Exceptions -2

Exceptions can be managed by ”catching” them:

file: `error1.py`

```
import error   <-- 1

def division(a,b):
    try:
        return error.division(a, b)   <-- 2
    except:
        print("You can't divide by zero!")   <-- 3
```

1. Note: a new way to import functions
2. If an exception happens here ...
3. .. do this

And here it is:

```
>>> from error1 import division
>>> division(2.33, 0)
You can't divide by zero!
>>>
```

Exceptions -2

Exceptions can be managed by ”catching” them:

file: `error1.py`

```
import error   <-- 1

def division(a,b):
    try:
        return error.division(a, b)   <-- 2
    except:
        print("You can't divide by zero!")   <-- 3
```

1. Note: a new way to import functions
2. If an exception happens here ...
3. .. do this

And here it is:

```
>>> from error1 import division
>>> division(2.33, 0)
You can't divide by zero!
>>>
```

Note: the exception mechanism allows the programmer to manage the error at the proper level.

E.g.: in `error1.py` the exception is catch in the caller of the function where the error happens.

The principle is: in case of error the exception climbs up the sequence of nested calls until catch. If it is not catch somewhere, the program terminates with the default behavior

Exceptions -2

Exceptions can be managed by ”catching” them:

file: `error1.py`

```
import error   <-- 1

def division(a,b):
    try:
        return error.division(a, b)   <-- 2
    except:
        print("You can't divide by zero!")   <-- 3
```

1. Note: a new way to import functions
2. If an exception happens here ...
3. .. do this

And here it is:

```
>>> from error1 import division
>>> division(2.33, 0)
You can't divide by zero!
>>>
```

Note: the exception mechanism allows the programmer to manage the error at the proper level.

E.g.: in `error1.py` the exception is catch in the caller of the function where the error happens.

The principle is: in case of error the exception climbs up the sequence of nested calls until catch. If it is not catch somewhere, the program terminates with the default behavior

→

Exceptions -3

Now let's try something different:

```
>>> division("two", 2)
You can't divide by zero!
>>>
```

Exceptions -3

Now let's try something different:

```
>>> division("two", 2)
You can't divide by zero!
>>>
```

Exceptions -3

Now let's try something different:

```
>>> division("two", 2)
You can't divide by zero!
>>>
```

← 1

1. Improper error management

Exceptions -3

Now let's try something different:

```
>>> division("two", 2)
You can't divide by zero!  ← 1
>>>
```

1. Improper error management

The exception mechanism provides for more detailed management:

Exceptions -3

Now let's try something different:

```
>>> division("two", 2)
You can't divide by zero!
>>>
```

← 1

1. Improper error management

The exception mechanism provides for more detailed management:

file: `error2.py`

```
import error

def division(a,b):
    try:
        return error.division(a, b)
    except ZeroDivisionError:
        print("You can't divide by zero!")
```

Exceptions -3

Now let's try something different:

```
>>> division("two", 2)
You can't divide by zero!  ← 1
>>>
```

1. Improper error management

The exception mechanism provides for more detailed management:

file: `error2.py`

```
import error

def division(a,b):
    try:
        return error.division(a, b)
    except ZeroDivisionError:  ← 1
        print("You can't divide by zero!")
```

Exceptions -3

Now let's try something different:

```
>>> division("two", 2)
You can't divide by zero!  ← 1
>>>
```

1. Improper error management

The exception mechanism provides for more detailed management:

file: `error2.py`

```
import error

def division(a,b):
    try:
        return error.division(a, b)
    except ZeroDivisionError:  ← 1
        print("You can't divide by zero!")
```

1. Catch only **ZeroDivisionError**

Exceptions -3

Now let's try something different:

```
>>> division("two", 2)
You can't divide by zero!  ← 1
>>>
```

1. Improper error management

The exception mechanism provides for more detailed management:

file: `error2.py`

```
import error

def division(a,b):
    try:
        return error.division(a, b)
    except ZeroDivisionError:  ← 1
        print("You can't divide by zero!")
```

1. Catch only **ZeroDivisionError**

→

Exceptions -4

Now everything works better:

```
>>> from error2 import division
>>> division(2,0)
You can't divide by zero!
>>> division("two",2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/error2.py", li
    return error.division(a, b)
  File "/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/error.py", lin
    return a/b
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Exceptions -4

Now everything works better:

```
>>> from error2 import division
>>> division(2,0)
You can't divide by zero!
>>> division("two",2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/error2.py", li
    return error.division(a, b)
  File "/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/error.py", lin
    return a/b
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

← 1

← 2

Exceptions -4

Now everything works better:

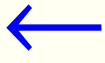
```
>>> from error2 import division
>>> division(2,0)
You can't divide by zero! ← 1
>>> division("two",2)
Traceback (most recent call last): ← 2
  File "<stdin>", line 1, in <module>
  File "/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/error2.py", li
    return error.division(a, b)
  File "/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/error.py", lin
    return a/b
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

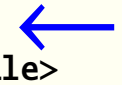
1. The exception `ZeroDivisionError` is catch and managed by the program

Exceptions -4

Now everything works better:

```
>>> from error2 import division
>>> division(2,0)
You can't divide by zero!
>>> division("two",2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/error2.py", li
    return error.division(a, b)
  File "/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/error.py", lin
    return a/b
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```





1. The exception `ZeroDivisionError` is catch and managed by the program
2. Any other exception is not managed, and causes the default program termination

Exceptions -4

Now everything works better:

```
>>> from error2 import division
>>> division(2,0)
You can't divide by zero!        <-- 1
>>> division("two",2)
Traceback (most recent call last):        <-- 2
  File "<stdin>", line 1, in <module>
  File "/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/error2.py", li
    return error.division(a, b)
  File "/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/error.py", lin
    return a/b
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

1. The exception `ZeroDivisionError` is catch and managed by the program

2. Any other exception is not managed, and causes the default program termination

Note 1: the `except` clause can provide also the catched exception with the following syntax:

```
except ZeroDivisionError as excp:
```

where the variable `excp` contains the exception object

Exceptions -4

Now everything works better:

```
>>> from error2 import division
>>> division(2,0)
You can't divide by zero!        <-- 1
>>> division("two",2)
Traceback (most recent call last):        <-- 2
  File "<stdin>", line 1, in <module>
  File "/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/error2.py", li
    return error.division(a, b)
  File "/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/error.py", lin
    return a/b
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

1. The exception `ZeroDivisionError` is catch and managed by the program

2. Any other exception is not managed, and causes the default program termination

Note 1: the `except` clause can provide also the catched exception with the following syntax:

```
except ZeroDivisionError as excp:
```

where the variable `excp` contains the exception object

Note 2: The list of predefined exceptions is in the manual (section: *Library Reference*)

Exceptions -4

Now everything works better:

```
>>> from error2 import division
>>> division(2,0)
You can't divide by zero!        <-- 1
>>> division("two",2)
Traceback (most recent call last):        <-- 2
  File "<stdin>", line 1, in <module>
  File "/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/error2.py", li
    return error.division(a, b)
  File "/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/error.py", lin
    return a/b
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

1. The exception `ZeroDivisionError` is catch and managed by the program

2. Any other exception is not managed, and causes the default program termination

Note 1: the `except` clause can provide also the catched exception with the following syntax:

```
except ZeroDivisionError as excp:
```

where the variable `excp` contains the exception object

Note 2: The list of predefined exceptions is in the manual (section: *Library Reference*)

→

Built-in functions

Python provides a number of functions for general use, they are referred to as `built-in functions`.

Built-in functions

Python provides a number of functions for general use, they are referred to as `built-in functions`.

Here follows the begin of the related section in the manual

2. Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Built-in functions

Python provides a number of functions for general use, they are referred to as `built-in functions`.

Here follows the begin of the related section in the manual

2. Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Many other functions are available in **standard modules**.

Built-in functions

Python provides a number of functions for general use, they are referred to as `built-in functions`.

Here follows the begin of the related section in the manual

2. Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Many other functions are available in **standard modules**.

→

Modules -1

Module: A block of code contained in a single file, to be used by other Python programs.

Modules -1

Module: A block of code contained in a single file, to be used by other Python programs.

file: `fibo.py`

```
"Module for the computation of Fibonacci series"

MAXFIBO=1000

def fibo(n):
    "Returns the Fibonacci series up to n"
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

if __name__=='__main__':
    print(fibo(34))
```

Modules -1

Module: A block of code contained in a single file, to be used by other Python programs.

file: `fibo.py`

```
"Module for the computation of Fibonacci series"   ← 1

MAXFIBO=1000   ← 2

def fibo(n):   ← 3
    "Returns the Fibonacci series up to n"   ← 4
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

if __name__=='__main__':   ← 5
    print(fibo(34))
```

Modules -1

Module: A block of code contained in a single file, to be used by other Python programs.

file: `fibo.py`

```
"Module for the computation of Fibonacci series"   ← 1

MAXFIBO=1000   ← 2

def fibo(n):   ← 3
    "Returns the Fibonacci series up to n"   ← 4
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

if __name__=='__main__':   ← 5
    print(fibo(34))
```

1. The module documentation string

Modules -1

Module: A block of code contained in a single file, to be used by other Python programs.

file: `fibo.py`

```
"Module for the computation of Fibonacci series"   ← 1

MAXFIBO=1000   ← 2

def fibo(n):   ← 3
    "Returns the Fibonacci series up to n"   ← 4
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

if __name__=='__main__':   ← 5
    print(fibo(34))
```

1. The module documentation string
2. A constant named: **MAXFIBO**

Modules -1

Module: A block of code contained in a single file, to be used by other Python programs.

file: `fibo.py`

```
"Module for the computation of Fibonacci series"   ← 1

MAXFIBO=1000   ← 2

def fibo(n):   ← 3
    "Returns the Fibonacci series up to n"   ← 4
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

if __name__=='__main__':   ← 5
    print(fibo(34))
```

1. The module documentation string
2. A constant named: **MAXFIBO**
3. A function named: **fibo()**

Modules -1

Module: A block of code contained in a single file, to be used by other Python programs.

file: `fibo.py`

```
"Module for the computation of Fibonacci series"   ← 1

MAXFIBO=1000   ← 2

def fibo(n):   ← 3
    "Returns the Fibonacci series up to n"   ← 4
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

if __name__=='__main__':   ← 5
    print(fibo(34))
```

1. The module documentation string
2. A constant named: **MAXFIBO**
3. A function named: **fibo()**
4. The function documentation string

Modules -1

Module: A block of code contained in a single file, to be used by other Python programs.

file: `fibo.py`

```
"Module for the computation of Fibonacci series"   <-- 1

MAXFIBO=1000   <-- 2

def fibo(n):   <-- 3
    "Returns the Fibonacci series up to n"   <-- 4
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

if __name__=='__main__':   <-- 5
    print(fibo(34))
```

1. The module documentation string
2. A constant named: **MAXFIBO**
3. A function named: **fibo()**
4. The function documentation string
5. Some test code (more about it in the following)

Modules -1

Module: A block of code contained in a single file, to be used by other Python programs.

file: `fibo.py`

```
"Module for the computation of Fibonacci series"   ← 1

MAXFIBO=1000   ← 2

def fibo(n):   ← 3
    "Returns the Fibonacci series up to n"   ← 4
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

if __name__=='__main__':   ← 5
    print(fibo(34))
```

1. The module documentation string
2. A constant named: **MAXFIBO**
3. A function named: **fibo()**
4. The function documentation string
5. Some test code (more about it in the following)

→

Using a module

```
>>> import fibo
>>> fibo.fibo(400)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
>>> fibo.MAXFIBO
1000
>>> dir(fibo)
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'fibo']
>>> fibo.__name__
'fibo'
>>> fibo.__doc__
'Module for the computation of Fibonacci series'
>>> fibo.__file__
'/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/fibo.py'
>>>
```

Modules -2

Using a module

```
>>> import fibo ← 1
>>> fibo.fibo(400) ← 2
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
>>> fibo.MAXFIBO ← 3
1000
>>> dir(fibo) ← 4
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'fibo']
>>> fibo.__name__ ← 5
'fibo'
>>> fibo.__doc__ ← 6
'Module for the computation of Fibonacci series'
>>> fibo.__file__ ← 7
'/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/fibo.py'
>>>
```

Modules -2

Using a module

```
>>> import fibo ← 1
>>> fibo.fibo(400) ← 2
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
>>> fibo.MAXFIBO ← 3
1000
>>> dir(fibo) ← 4
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'fibo']
>>> fibo.__name__ ← 5
'fibo'
>>> fibo.__doc__ ← 6
'Module for the computation of Fibonacci series'
>>> fibo.__file__ ← 7
'/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/fibo.py'
>>>
```

1. A module must be **imported**

Modules -2

Using a module

```
>>> import fibo ← 1
>>> fibo.fibo(400) ← 2
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
>>> fibo.MAXFIBO ← 3
1000
>>> dir(fibo) ← 4
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'fibo']
>>> fibo.__name__ ← 5
'fibo'
>>> fibo.__doc__ ← 6
'Module for the computation of Fibonacci series'
>>> fibo.__file__ ← 7
'/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/fibo.py'
>>>
```

1. A module must be **imported**
2. How to call the module's **fibo()** function

Modules -2

Using a module

```
>>> import fibo ← 1
>>> fibo.fibo(400) ← 2
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
>>> fibo.MAXFIBO ← 3
1000
>>> dir(fibo) ← 4
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'fibo']
>>> fibo.__name__ ← 5
'fibo'
>>> fibo.__doc__ ← 6
'Module for the computation of Fibonacci series'
>>> fibo.__file__ ← 7
'/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/fibo.py'
>>>
```

1. A module must be **imported**
2. How to call the module's **fibo()** function
3. Getting variable **MAXFIBO** from the module

Modules -2

Using a module

```
>>> import fibo ← 1
>>> fibo.fibo(400) ← 2
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
>>> fibo.MAXFIBO ← 3
1000
>>> dir(fibo) ← 4
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'fibo']
>>> fibo.__name__ ← 5
'fibo'
>>> fibo.__doc__ ← 6
'Module for the computation of Fibonacci series'
>>> fibo.__file__ ← 7
'/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/fibo.py'
>>>
```

1. A module must be **imported**
2. How to call the module's **fibo()** function
3. Getting variable **MAXFIBO** from the module
4. The built-in function **dir()** shows module's content, including *internal variables*

Modules -2

Using a module

```
>>> import fibo ← 1
>>> fibo.fibo(400) ← 2
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
>>> fibo.MAXFIBO ← 3
1000
>>> dir(fibo) ← 4
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'fibo']
>>> fibo.__name__ ← 5
'fibo'
>>> fibo.__doc__ ← 6
'Module for the computation of Fibonacci series'
>>> fibo.__file__ ← 7
'/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/fibo.py'
>>>
```

1. A module must be **imported**
2. How to call the module's **fibo()** function
3. Getting variable **MAXFIBO** from the module
4. The built-in function **dir()** shows module's content, including *internal variables*
5. The internal variable **__name__** holds the module's name (see also next slide)

Modules -2

Using a module

```
>>> import fibo ← 1
>>> fibo.fibo(400) ← 2
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
>>> fibo.MAXFIBO ← 3
1000
>>> dir(fibo) ← 4
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'fibo']
>>> fibo.__name__ ← 5
'fibo'
>>> fibo.__doc__ ← 6
'Module for the computation of Fibonacci series'
>>> fibo.__file__ ← 7
'/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/fibo.py'
>>>
```

1. A module must be **imported**
2. How to call the module's **fibo()** function
3. Getting variable **MAXFIBO** from the module
4. The built-in function **dir()** shows module's content, including *internal variables*
5. The internal variable **__name__** holds the module's name (see also next slide)
6. The internal variable **__doc__** holds the *documentation string*

Modules -2

Using a module

```
>>> import fibo ← 1
>>> fibo.fibo(400) ← 2
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
>>> fibo.MAXFIBO ← 3
1000
>>> dir(fibo) ← 4
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'fibo']
>>> fibo.__name__ ← 5
'fibo'
>>> fibo.__doc__ ← 6
'Module for the computation of Fibonacci series'
>>> fibo.__file__ ← 7
'/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/fibo.py'
>>>
```

1. A module must be **imported**
2. How to call the module's **fibo()** function
3. Getting variable **MAXFIBO** from the module
4. The built-in function **dir()** shows module's content, including *internal variables*
5. The internal variable **__name__** holds the module's name (see also next slide)
6. The internal variable **__doc__** holds the *documentation string*
7. The internal variable **__file__** holds the source file path

Modules -2

Using a module

```
>>> import fibo ← 1
>>> fibo.fibo(400) ← 2
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
>>> fibo.MAXFIBO ← 3
1000
>>> dir(fibo) ← 4
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'fibo']
>>> fibo.__name__ ← 5
'fibo'
>>> fibo.__doc__ ← 6
'Module for the computation of Fibonacci series'
>>> fibo.__file__ ← 7
'/home/lfini/Personale/CorsiSeminari/2019-Python/intro_code/fibo.py'
>>>
```

1. A module must be **imported**
2. How to call the module's **fibo()** function
3. Getting variable **MAXFIBO** from the module
4. The built-in function **dir()** shows module's content, including *internal variables*
5. The internal variable **__name__** holds the module's name (see also next slide)
6. The internal variable **__doc__** holds the *documentation string*
7. The internal variable **__file__** holds the source file path

→

Modules -3

When a module is imported for the first time, its code is executed and is made available to the importer.

If the module is imported again during the execution of the same program the code is **not** re-executed

Modules -3

When a module is imported for the first time, its code is executed and is made available to the importer.

If the module is imported again during the execution of the same program the code is **not** re-executed

More about the module's variable `__name__`

Modules -3

When a module is imported for the first time, its code is executed and is made available to the importer.

If the module is imported again during the execution of the same program the code is **not** re-executed

More about the module's variable `__name__`

- Variable `__name__` holds the module's name (`"fibo"` in the example), but **only** when the module is imported

Modules -3

When a module is imported for the first time, its code is executed and is made available to the importer.

If the module is imported again during the execution of the same program the code is **not** re-executed

More about the module's variable `__name__`

- Variable `__name__` holds the module's name (`"fibo"` in the example), but **only** when the module is imported
- When the module file is executed as a program `__name__` holds the string `"__main__"`

Modules -3

When a module is imported for the first time, its code is executed and is made available to the importer.

If the module is imported again during the execution of the same program the code is **not** re-executed

More about the module's variable `__name__`

- Variable `__name__` holds the module's name (`"fibo"` in the example), but **only** when the module is imported
- When the module file is executed as a program `__name__` holds the string `"__main__"`
- As a consequence when the module is imported, the code under the conditional (`if __name__=="__main__":`) is not executed.

Modules -3

When a module is imported for the first time, its code is executed and is made available to the importer.

If the module is imported again during the execution of the same program the code is **not** re-executed

More about the module's variable `__name__`

- Variable `__name__` holds the module's name (`"fibo"` in the example), but **only** when the module is imported
- When the module file is executed as a program `__name__` holds the string `"__main__"`
- As a consequence when the module is imported, the code under the conditional (`if __name__=="__main__":`) is not executed.
- It is common practice to put there test code intended to be executed only by direct invocation

Modules -3

When a module is imported for the first time, its code is executed and is made available to the importer.

If the module is imported again during the execution of the same program the code is **not** re-executed

More about the module's variable `__name__`

- Variable `__name__` holds the module's name (`"fibo"` in the example), but **only** when the module is imported
- When the module file is executed as a program `__name__` holds the string `"__main__"`
- As a consequence when the module is imported, the code under the conditional (`if __name__=="__main__":`) is not executed.
- It is common practice to put there test code intended to be executed only by direct invocation

→

namespaces and *scope* -1

namespaces and *scope* -1

- A *namespace* is a “container” for names.

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py`

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):
    cube = n*n*n
    print(note % (n, cube))
```

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py` ← 1

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):          ← 2,3
    cube = n*n*n
    print(note % (n, cube))
```

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py` ← 1

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n): ← 2,3
    cube = n*n*n
    print(note % (n, cube))
```

1. Module `prcube` is a namespace containing names: **note**, **cube**, **print_cube**

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py` ← 1

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):          ← 2,3
    cube = n*n*n
    print(note % (n, cube))
```

1. Module `prcube` is a namespace containing names: **note**, **cube**, **print_cube**
2. The function `print_cube()` is another namespace containing names **cube** and **n**

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py` ← 1

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):          ← 2,3
    cube = n*n*n
    print(note % (n, cube))
```

1. Module `prcube` is a namespace containing names: **note**, **cube**, **print_cube**
2. The function `print_cube()` is another namespace containing names **cube** and **n**
3. **Note:** the local namespace of any function is not visible outside the function

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py` ← 1

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):          ← 2,3
    cube = n*n*n
    print(note % (n, cube))
```

1. Module `prcube` is a namespace containing names: **note**, **cube**, **print_cube**
2. The function `print_cube()` is another namespace containing names **cube** and **n**
3. **Note:** the local namespace of any function is not visible outside the function

→

namespaces and *scope* -2

Let's experiment with the Python interpreter:

namespaces and *scope* -2

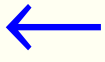
Let's experiment with the Python interpreter:

```
$ python
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> cube = "The third power"

>>> import prcube

>>> prcube.print_cube(11)
The cube of 11 is: 1331

>>> cube
'The third power'

>>> prcube.cube
3.1415926
```

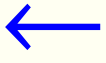
namespaces and *scope* -2

Let's experiment with the Python interpreter:

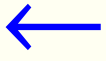
```
$ python
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> cube = "The third power"
```




```
>>> import prcube
```




```
>>> prcube.print_cube(11)
```




```
The cube of 11 is: 1331

>>> cube
'The third power'

>>> prcube.cube
3.1415926
```

namespaces and *scope* -2

Let's experiment with the Python interpreter:

```
$ python
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> cube = "The third power"  <-- 1

>>> import prcube  <-- 2

>>> prcube.print_cube(11)  <-- 3
The cube of 11 is: 1331

>>> cube
'The third power'

>>> prcube.cube
3.1415926
```

1. The name **cube** is created in level 0 namespace (the Python interpreter)

namespaces and *scope* -2

Let's experiment with the Python interpreter:

```
$ python
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> cube = "The third power"  <-- 1

>>> import prcube  <-- 2

>>> prcube.print_cube(11)  <-- 3
The cube of 11 is: 1331

>>> cube
'The third power'

>>> prcube.cube
3.1415926
```

1. The name **cube** is created in level 0 namespace (the Python interpreter)
2. The **import** statement creates the name **prcube** referring to the full namespace of the `prcube` module, which is nested within the level 0 namespace.

namespaces and *scope* -2

Let's experiment with the Python interpreter:

```
$ python
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> cube = "The third power"  <-- 1

>>> import prcube  <-- 2

>>> prcube.print_cube(11)  <-- 3
The cube of 11 is: 1331

>>> cube
'The third power'

>>> prcube.cube
3.1415926
```

1. The name **cube** is created in level 0 namespace (the Python interpreter)
2. The **import** statement creates the name **prcube** referring to the full namespace of the `prcube` module, which is nested within the level 0 namespace.
3. The **print_cube()** function defines another namespace, nested within the `prcube` namespace, containing a variable named **cube**

namespaces and *scope* -2

Let's experiment with the Python interpreter:

```
$ python
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> cube = "The third power"  <-- 1

>>> import prcube  <-- 2

>>> prcube.print_cube(11)  <-- 3
The cube of 11 is: 1331

>>> cube
'The third power'

>>> prcube.cube
3.1415926
```

1. The name **cube** is created in level 0 namespace (the Python interpreter)
2. The **import** statement creates the name **prcube** referring to the full namespace of the `prcube` module, which is nested within the level 0 namespace.
3. The **print_cube()** function defines another namespace, nested within the `prcube` namespace, containing a variable named **cube**

The three variables named **cube** do not conflict because they belong to different namespaces.

namespaces and *scope* -3

- ***Scope***: All the “area” where a name is referred to the same object.

namespaces and *scope* -3

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:

namespaces and *scope* -3

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  1. Name creation: in the currently active namespace.

namespaces and *scope* -3

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  1. Name creation: in the currently active namespace.
  2. Name search: starting from currently active namespace and up the hierarchy

namespaces and *scope* -3

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  1. Name creation: in the currently active namespace.
  2. Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the previous example:

```
note = "The cube of %d is: %d"
cube = 3.1415926
def print_cube(n):
    cube = n*n*n
    print(note % (n, cube))
....
```

namespaces and *scope* -3

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  1. Name creation: in the currently active namespace.
  2. Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the previous example:

```
note = "The cube of %d is: %d"  <-- 1
cube = 3.1415926  <-- 2
def print_cube(n):  <-- 3
    cube = n*n*n  <-- 4
    print(note % (n, cube))  <-- 5
....  <-- 6
```

namespaces and *scope* -3

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  1. Name creation: in the currently active namespace.
  2. Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the previous example:

```
note = "The cube of %d is: %d"  <-- 1
cube = 3.1415926                <-- 2
def print_cube(n):              <-- 3
    cube = n*n*n                <-- 4
    print(note % (n, cube))     <-- 5
....                            <-- 6
```

1. The name **note** is created here

namespaces and *scope* -3

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  1. Name creation: in the currently active namespace.
  2. Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the previous example:

```
note = "The cube of %d is: %d"  <-- 1
cube = 3.1415926                <-- 2
def print_cube(n):              <-- 3
    cube = n*n*n                <-- 4
    print(note % (n, cube))     <-- 5
....                            <-- 6
```

1. The name **note** is created here
2. The name **cube**$_1$ is created here

namespaces and *scope* -3

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  1. Name creation: in the currently active namespace.
  2. Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the previous example:

```
note = "The cube of %d is: %d"  <-- 1
cube = 3.1415926                <-- 2
def print_cube(n):              <-- 3
    cube = n*n*n                <-- 4
    print(note % (n, cube))     <-- 5
....                            <-- 6
```

1. The name **note** is created here
2. The name **cube**$_1$ is created here
3. The name **n** is created here

namespaces and *scope* -3

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  1. Name creation: in the currently active namespace.
  2. Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the previous example:

```
note = "The cube of %d is: %d"  <-- 1
cube = 3.1415926                <-- 2
def print_cube(n):              <-- 3
    cube = n*n*n                <-- 4
    print(note % (n, cube))     <-- 5
....                            <-- 6
```

1. The name **note** is created here
2. The name **cube**$_1$ is created here
3. The name **n** is created here
4. The name **cube**$_2$ is created here and hides **cube**$_1$

namespaces and *scope* -3

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  1. Name creation: in the currently active namespace.
  2. Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the previous example:

```
note = "The cube of %d is: %d"  <-- 1
cube = 3.1415926                <-- 2
def print_cube(n):              <-- 3
    cube = n*n*n                <-- 4
    print(note % (n, cube))     <-- 5
....                            <-- 6
```

1. The name **note** is created here
2. The name **cube**$_1$ is created here
3. The name **n** is created here
4. The name **cube**$_2$ is created here and hides **cube**$_1$
5. Here the name **note** is searched up the hierarchy and found one step above. **n** and **cube** are found in the current namespace

namespaces and *scope* -3

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  1. Name creation: in the currently active namespace.
  2. Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the previous example:

```
note = "The cube of %d is: %d"  <-- 1
cube = 3.1415926                <-- 2
def print_cube(n):              <-- 3
    cube = n*n*n                <-- 4
    print(note % (n, cube))     <-- 5
....                            <-- 6
```

1. The name **note** is created here
2. The name **cube**$_1$ is created here
3. The name **n** is created here
4. The name **cube**$_2$ is created here and hides **cube**$_1$
5. Here the name **note** is searched up the hierarchy and found one step above. **n** and **cube** are found in the current namespace
6. Here the name **cube**$_2$ disappears and **cube** refers again to the object created at the second line of the file

namespaces and *scope* -3

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  1. Name creation: in the currently active namespace.
  2. Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the previous example:

```
note = "The cube of %d is: %d"  <-- 1
cube = 3.1415926                <-- 2
def print_cube(n):              <-- 3
    cube = n*n*n                <-- 4
    print(note % (n, cube))     <-- 5
....                            <-- 6
```

1. The name **note** is created here
2. The name **cube**$_1$ is created here
3. The name **n** is created here
4. The name **cube**$_2$ is created here and hides **cube**$_1$
5. Here the name **note** is searched up the hierarchy and found one step above. **n** and **cube** are found in the current namespace
6. Here the name **cube**$_2$ disappears and **cube** refers again to the object created at the second line of the file

namespaces and *scope* -4

Let's start with a very simple little program which prints the value of a variable if that value is greather than zero:

namespaces and *scope* -4

Let's start with a very simple little program which prints the value of a variable if that value is greather than zero:

file: `scope1.py`

```
CURRENT_DISK = 3

def new_disk():
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)

if __name__ == "__main__":
    new_disk()
```

namespaces and *scope* -4

Let's start with a very simple little program which prints the value of a variable if that value is greather than zero:

file: `scope1.py`

```
CURRENT_DISK = 3

def new_disk():          <-- 1
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)

if __name__ == "__main__":
    new_disk()           <-- 1
```

namespaces and *scope* -4

Let's start with a very simple little program which prints the value of a variable if that value is greather than zero:

file: `scope1.py`

```
CURRENT_DISK = 3

def new_disk():  <-- 1
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)

if __name__ == "__main__":
    new_disk()  <-- 1
```

1. Incidentally note that the function name `main()` has nothing special

namespaces and *scope* -4

Let's start with a very simple little program which prints the value of a variable if that value is greather than zero:

file: `scope1.py`

```
CURRENT_DISK = 3

def new_disk():  <-- 1
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)

if __name__ == "__main__":
    new_disk()  <-- 1
```

1. Incidentally note that the function name `main()` has nothing special

Let's run the above script:

```
$: python scope1.py
Current disk: 3
```

namespaces and *scope* -4

Let's start with a very simple little program which prints the value of a variable if that value is greather than zero:

file: `scope1.py`

```
CURRENT_DISK = 3

def new_disk():  <-- 1
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)

if __name__ == "__main__":
    new_disk()  <-- 1
```

1. Incidentally note that the function name `main()` has nothing special

Let's run the above script:

```
$: python scope1.py
Current disk: 3
```

The programs does what's expected (nothing really useful anyway ...)

namespaces and *scope* -4

Let's start with a very simple little program which prints the value of a variable if that value is greather than zero:

file: `scope1.py`

```
CURRENT_DISK = 3

def new_disk():  <-- 1
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)

if __name__ == "__main__":
    new_disk()  <-- 1
```

1. Incidentally note that the function name `main()` has nothing special

Let's run the above script:

```
$: python scope1.py
Current disk: 3
```

The programs does what's expected (nothing really useful anyway ...)

→

namespaces and *scope* -5

Let's now suppose we also want to change the value of `CURRENT_DISK` variable when the procedure is called...

namespaces and *scope* -5

Let's now suppose we also want to change the value of `CURRENT_DISK` variable when the procedure is called...

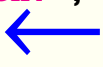
file: `scope2.py`

```
CURRENT_DISK = 3

def new_disk():
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)
        CURRENT_DISK = 5

if __name__ == "__main__":
    new_disk()
```

namespaces and *scope* -5

Let's now suppose we also want to change the value of `CURRENT_DISK` variable when the procedure is called...

file: `scope2.py`

```
CURRENT_DISK = 3

def new_disk():
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)
        CURRENT_DISK = 5



if __name__ == "__main__":
    new_disk()
```

namespaces and *scope* -5

Let's now suppose we also want to change the value of `CURRENT_DISK` variable when the procedure is called...

file: `scope2.py`

```
CURRENT_DISK = 3

def new_disk():
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)
        CURRENT_DISK = 5  <-- 1

if __name__ == "__main__":
    new_disk()
```

1. I've simply added an assignment statement

namespaces and *scope* -5

Let's now suppose we also want to change the value of `CURRENT_DISK` variable when the procedure is called...

file: `scope2.py`

```
CURRENT_DISK = 3

def new_disk():
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)
        CURRENT_DISK = 5  <-- 1

if __name__ == "__main__":
    new_disk()
```

1. I've simply added an assignment statement

Let's run the new version:

```
$ python scope2.py
Traceback (most recent call last):
  File "scope2.py", line 9, in <module>
    new_disk()
  File "scope2.py", line 4, in new_disk
    if CURRENT_DISK:
UnboundLocalError: local variable 'CURRENT_DISK' referenced before assignment
```

namespaces and *scope* -5

Let's now suppose we also want to change the value of `CURRENT_DISK` variable when the procedure is called...

file: `scope2.py`

```
CURRENT_DISK = 3

def new_disk():
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)
        CURRENT_DISK = 5  <-- 1

if __name__ == "__main__":
    new_disk()
```

1. I've simply added an assignment statement

Let's run the new version:

```
$ python scope2.py
Traceback (most recent call last):
  File "scope2.py", line 9, in <module>
    new_disk()
  File "scope2.py", line 4, in new_disk
    if CURRENT_DISK:
UnboundLocalError: local variable 'CURRENT_DISK' referenced before assignment
```

What happened?

Let's now suppose we also want to change the value of `CURRENT_DISK` variable when the procedure is called...

file: `scope2.py`

```
CURRENT_DISK = 3

def new_disk():
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)
        CURRENT_DISK = 5  <-- 1

if __name__ == "__main__":
    new_disk()
```

1. I've simply added an assignment statement

Let's run the new version:

```
$ python scope2.py
Traceback (most recent call last):
  File "scope2.py", line 9, in <module>
    new_disk()
  File "scope2.py", line 4, in new_disk
    if CURRENT_DISK:
UnboundLocalError: local variable 'CURRENT_DISK' referenced before assignment
```

What happened?

→

Let's go back to scope rules:

Let's go back to scope rules:

- In `scope2.py`, at line 5, we have an assignment: `CURRENT_DISK = 5`, i.e.: a name creation

Let's go back to scope rules:

- In `scope2.py`, at line 5, we have an assignment: `CURRENT_DISK = 5`, i.e.: a name creation
- Thus the name `CURRENT_DISK` is to be located in the **local** namespace of function `new_disk()` due to scope rule 1

Let's go back to scope rules:

- In `scope2.py`, at line 5, we have an assignment: `CURRENT_DISK = 5`, i.e.: a name creation
- Thus the name `CURRENT_DISK` is to be located in the **local** namespace of function `new_disk()` due to scope rule 1
- But the variable is actually created **only when** the assignment statement is executed; i.e.: when the `if` statement is executed the variable has not yet been created.

Let's go back to scope rules:

- In `scope2.py`, at line 5, we have an assignment: `CURRENT_DISK = 5`, i.e.: a name creation
- Thus the name `CURRENT_DISK` is to be located in the **local** namespace of function `new_disk()` due to scope rule 1
- But the variable is actually created **only when** the assignment statement is executed; i.e.: when the `if` statement is executed the variable has not yet been created.
- In other terms: the variable `CURRENT_DISK` at line 5 and the variable `CURRENT_DISK` at line 1 are different variables!

namespaces and *scope* -6

Let's go back to scope rules:

- In `scope2.py`, at line 5, we have an assignment: `CURRENT_DISK = 5`, i.e.: a name creation
- Thus the name `CURRENT_DISK` is to be located in the **local** namespace of function `new_disk()` due to scope rule 1
- But the variable is actually created **only when** the assignment statement is executed; i.e.: when the `if` statement is executed the variable has not yet been created.
- In other terms: the variable `CURRENT_DISK` at line 5 and the variable `CURRENT_DISK` at line 1 are different variables!

- In `scope1.py`, there is no assignment to `CURRENT_DISK`, so scope rule 2 holds.

Let's go back to scope rules:

- In `scope2.py`, at line 5, we have an assignment: `CURRENT_DISK = 5`, i.e.: a name creation
- Thus the name `CURRENT_DISK` is to be located in the **local** namespace of function `new_disk()` due to scope rule 1
- But the variable is actually created **only when** the assignment statement is executed; i.e.: when the `if` statement is executed the variable has not yet been created.
- In other terms: the variable `CURRENT_DISK` at line 5 and the variable `CURRENT_DISK` at line 1 are different variables!

- In `scope1.py`, there is no assignment to `CURRENT_DISK`, so scope rule 2 holds.

→

global namespace -1

global namespace -1

file: `scope3.py` – Working version

```
CURRENT_DISK = 3

def new_disk():
    global CURRENT_DISK
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)
        CURRENT_DISK = 5

if __name__ == "__main__":
    new_disk()
```

global namespace -1

file: `scope3.py` – Working version

```
CURRENT_DISK = 3

def new_disk():
    global CURRENT_DISK
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)
        CURRENT_DISK = 5

if __name__ == "__main__":
    new_disk()
```

global namespace -1

file: `scope3.py` – Working version

```
CURRENT_DISK = 3

def new_disk():
    global CURRENT_DISK <-- 1
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)
        CURRENT_DISK = 5

if __name__ == "__main__":
    new_disk()
```

1. The statement `global CURRENT_DISK` forces Python to create (or search) the name `CURRENT_DISK` into the global namespace, i.e.: the topmost in the namespace hierarchy.

global namespace -1

file: scope3.py – Working version

```
CURRENT_DISK = 3

def new_disk():
    global CURRENT_DISK <-- 1
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)
        CURRENT_DISK = 5

if __name__ == "__main__":
    new_disk()
```

1. The statement `global CURRENT_DISK` forces Python to create (or search) the name `CURRENT_DISK` into the global namespace, i.e.: the topmost in the namespace hierarchy.

Now we can run it

```
$ python scope3.py
Current disk: 3
```

global namespace -1

file: scope3.py – Working version

```
CURRENT_DISK = 3

def new_disk():
    global CURRENT_DISK ← 1
    if CURRENT_DISK:
        print("Current disk:", CURRENT_DISK)
        CURRENT_DISK = 5

if __name__ == "__main__":
    new_disk()
```

1. The statement `global CURRENT_DISK` forces Python to create (or search) the name `CURRENT_DISK` into the global namespace, i.e.: the topmost in the namespace hierarchy.

Now we can run it

```
$ python scope3.py
Current disk: 3
```

→

Packages

Python's packages are hierarchical structures gathering together several modules

Packages

Python's packages are hierarchical structures gathering together several modules

- Whenever a module becomes too complex, it could be usefully divided into sub-modules

Packages

Python's packages are hierarchical structures gathering together several modules

- Whenever a module becomes too complex, it could be usefully divided into sub-modules
- Sub-modules can be arranged hyerarchically into a **package**

Packages

Python's packages are hierarchical structures gathering together several modules

- Whenever a module becomes too complex, it could be usefully divided into sub-modules
- Sub-modules can be arranged hyerarchically into a **package**
- A package is a directory containing a file named `__init__.py`, the sub-modules Python files and, possibly, subdirectories.

Packages

Python's packages are hierarchical structures gathering together several modules

- Whenever a module becomes too complex, it could be usefully divided into sub-modules
- Sub-modules can be arranged hyerarchically into a **package**
- A package is a directory containing a file named **`__init__.py`**, the sub-modules Python files and, possibly, subdirectories.
- The file **`__init__.py`** may contain initialization code (or even be empty)

Packages

Python's packages are hierarchical structures gathering together several modules

- Whenever a module becomes too complex, it could be usefully divided into sub-modules
- Sub-modules can be arranged hyerarchically into a **package**
- A package is a directory containing a file named `__init__.py`, the sub-modules Python files and, possibly, subdirectories.
- The file `__init__.py` may contain initialization code (or even be empty)
- Packages can be imported as a whole, or you can import only selected sub-modules

Packages

Python's packages are hierarchical structures gathering together several modules

- Whenever a module becomes too complex, it could be usefully divided into sub-modules
- Sub-modules can be arranged hyerarchically into a **package**
- A package is a directory containing a file named `__init__.py`, the sub-modules Python files and, possibly, subdirectories.
- The file `__init__.py` may contain initialization code (or even be empty)
- Packages can be imported as a whole, or you can import only selected sub-modules
- We'll see packages in action in many followjng examples

Packages

Python's packages are hierarchical structures gathering together several modules

- Whenever a module becomes too complex, it could be usefully divided into sub-modules
- Sub-modules can be arranged hyerarchically into a **package**
- A package is a directory containing a file named `__init__.py`, the sub-modules Python files and, possibly, subdirectories.
- The file `__init__.py` may contain initialization code (or even be empty)
- Packages can be imported as a whole, or you can import only selected sub-modules
- We'll see packages in action in many followjng examples

→

Standard Modules and Packages

Standard Modules and Packages

- Python is particularly rich in modules and packages distributed together with the language interpreter.

Standard Modules and Packages

- Python is particularly rich in modules and packages distributed together with the language interpreter.
- The first thing to do when starting a new program is: look for a module or package which solves your problem

Standard Modules and Packages

- Python is particularly rich in modules and packages distributed together with the language interpreter.
- The first thing to do when starting a new program is: look for a module or package which solves your problem
- Standard modules:
  - sys
  - os
  - os.path
  - math, cmath
  - random
  - ... plus 280, more or less

Standard Modules and Packages

- Python is particularly rich in modules and packages distributed together with the language interpreter.
- The first thing to do when starting a new program is: look for a module or package which solves your problem
- Standard modules:
  - sys
  - os
  - os.path
  - math, cmath
  - random
  - ... plus 280, more or less
- Special interest packages:
  - numpy
  - scipy
  - matplotlib
  - astropy

Standard Modules and Packages

- Python is particularly rich in modules and packages distributed together with the language interpreter.
- The first thing to do when starting a new program is: look for a module or package which solves your problem
- Standard modules:
  - sys
  - os
  - os.path
  - math, cmath
  - random
  - ... plus 280, more or less
- Special interest packages:
  - numpy
  - scipy
  - matplotlib
  - astropy

→

The `sys` standard module

Classes, functions and constants directly related with the `Python` interpreter.

The sys standard module

Classes, functions and constants directly related with the Python interpreter.

Let's explore at the Python prompt:

- sys.path (See also: PYTHONPATH)
- sys.argv
- sys.exit()

The sys standard module

Classes, functions and constants directly related with the Python interpreter.

Let's explore at the Python prompt:

- sys.path (See also: PYTHONPATH)
- sys.argv
- sys.exit()
- sys.maxint
- sys.float_info
- sys.maxsize
- sys.platform

The sys standard module

Classes, functions and constants directly related with the Python interpreter.

Let's explore at the Python prompt:

- sys.path (See also: PYTHONPATH)
- sys.argv
- sys.exit()
- sys.maxint
- sys.float_info
- sys.maxsize
- sys.platform
- sys.stdin
- sys.stdout
- sys.stderr

The sys standard module

Classes, functions and constants directly related with the Python interpreter.

Let's explore at the Python prompt:

- sys.path (See also: PYTHONPATH)
- sys.argv
- sys.exit()
- sys.maxint
- sys.float_info
- sys.maxsize
- sys.platform
- sys.stdin
- sys.stdout
- sys.stderr

The os standard package -1

Classes, functions and constants directly related with the Operating System.

The os standard package -1

Classes, functions and constants directly related with the Operating System.

Let's explore at the Python prompt:

- os.sep
- os.linesep
- os.defpath
- os.environ
- os.getenv("HOME")
- os.curdir

The os standard package -1

Classes, functions and constants directly related with the Operating System.

Let's explore at the Python prompt:

- os.sep
- os.linesep
- os.defpath
- os.environ
- os.getenv("HOME")
- os.curdir

Some examples:

```
>>> import os
>>> os.sep
'/'
>>> os.defpath
':/bin:/usr/bin'
>>> os.environ
environ({'LC_MEASUREMENT': 'it_IT.UTF-8', 'DISPLAY': ':0.0', 'EDITOR': 'vim',
   ...})
>>> os.getenv("HOME")
'/home/lfini'
>>> os.environ["HOME"]
'/home/lfini'
>>> os.curdir
'.'
```

The os standard package -1

Classes, functions and constants directly related with the Operating System.

Let's explore at the Python prompt:

- os.sep
- os.linesep
- os.defpath
- os.environ
- os.getenv("HOME")
- os.curdir

Some examples:

```
>>> import os
>>> os.sep
'/'
>>> os.defpath
':/bin:/usr/bin'
>>> os.environ
environ({'LC_MEASUREMENT': 'it_IT.UTF-8', 'DISPLAY': ':0.0', 'EDITOR': 'vim',
   ...})
>>> os.getenv("HOME")
'/home/lfini'
>>> os.environ["HOME"]
'/home/lfini'
>>> os.curdir
'.'
```

→

The os standard package -2

Let's explore at the Python prompt:

- os.access("a.py",os.R_OK)
- os.chdir("newdir")
- os.listdir("dirname")
- os.mkdir("/dir1/dir2/name")
- os.makedirs("/dir1/dir2/name")
- os.rename("old","new")
- os.renames("old","new")
- os.walk("topdir")

The os standard package -2

Let's explore at the Python prompt:

- os.access("a.py",os.R_OK)
- os.chdir("newdir")
- os.listdir("dirname")
- os.mkdir("/dir1/dir2/name")
- os.makedirs("/dir1/dir2/name")
- os.rename("old","new")
- os.renames("old","new")
- os.walk("topdir")

More examples:

```
>>> os.access("a.py",os.R_OK)
False
>>> tree=os.walk("code")
>>> for dp,dnames,fnames in tree:
...   for fn in fnames:
...     print(os.path.join(dp,fn))
...
code/scope3.py
code/prcube.py
code/decorator.py
    ....
code/data/img-611.fit
code/data/img-646.fit
    ....
```

The os standard package -2

Let's explore at the Python prompt:

- os.access("a.py",os.R_OK)
- os.chdir("newdir")
- os.listdir("dirname")
- os.mkdir("/dir1/dir2/name")
- os.makedirs("/dir1/dir2/name")
- os.rename("old","new")
- os.renames("old","new")
- os.walk("topdir")

More examples:

```
>>> os.access("a.py",os.R_OK)
False
>>> tree=os.walk("code")
>>> for dp,dnames,fnames in tree:
...   for fn in fnames:
...     print(os.path.join(dp,fn))
...
code/scope3.py
code/prcube.py
code/decorator.py
    ....
code/data/img-611.fit
code/data/img-646.fit
    ....
```

→

The `os.path` sub-module

Functions to manipulate file names and paths in a portable way

The `os.path` sub-module

Functions to manipulate file names and paths in a portable way

Let's explore at the `Python` prompt:

- os.path.abspath(path)
- os.path.basename(path)
- os.path.dirname(path)
- os.path.split(path)
- os.path.commonprefix(list)
- os.path.exists(path)
- os.path.getatime(path)
- os.path.getmtime(path)
- os.path.getctime(path)
- os.path.join(path1,path2,..)

The `os.path` sub-module

Functions to manipulate file names and paths in a portable way

Let's explore at the `Python` prompt:

- os.path.abspath(path)
- os.path.basename(path)
- os.path.dirname(path)
- os.path.split(path)
- os.path.commonprefix(list)
- os.path.exists(path)
- os.path.getatime(path)
- os.path.getmtime(path)
- os.path.getctime(path)
- os.path.join(path1,path2,..)

→

Modules: `math`, `cmath`, `random`

Functions on real numbers (`math`) complex numbers (`cmath`) and random numbers generation (`random`)

Modules: `math`, `cmath`, `random`

Functions on real numbers (`math`) complex numbers (`cmath`) and random numbers generation (`random`)

Let's explore at the `Python` prompt:

- help(math), to be noted:
  - fsum
  - expm1
  - log1p

Modules: `math`, `cmath`, `random`

Functions on real numbers (`math`) complex numbers (`cmath`) and random numbers generation (`random`)

Let's explore at the `Python` prompt:

- help(math), to be noted:
  - fsum
  - expm1
  - log1p
- help(cmath)

Modules: `math`, `cmath`, `random`

Functions on real numbers (`math`) complex numbers (`cmath`) and random numbers generation (`random`)

Let's explore at the `Python` prompt:

- help(math), to be noted:
  - fsum
  - expm1
  - log1p
- help(cmath)
- help(random)

Modules: `math`, `cmath`, `random`

Functions on real numbers (`math`) complex numbers (`cmath`) and random numbers generation (`random`)

Let's explore at the `Python` prompt:

- help(math), to be noted:
  - fsum
  - expm1
  - log1p
- help(cmath)
- help(random)

→

Example 1: Sending e-mail

Sending mail messages from a procedure may be useful in many cases:

Example 1: Sending e-mail

Sending mail messages from a procedure may be useful in many cases:

- To manage mailing lists

Example 1: Sending e-mail

Sending mail messages from a procedure may be useful in many cases:

- To manage mailing lists
- To send alert messages from scripts

Example 1: Sending e-mail

Sending mail messages from a procedure may be useful in many cases:

- To manage mailing lists
- To send alert messages from scripts
- To send error messages to a program's development team

Example 1: Sending e-mail

Sending mail messages from a procedure may be useful in many cases:

- To manage mailing lists
- To send alert messages from scripts
- To send error messages to a program's development team
- ...

Example 1: Sending e-mail

Sending mail messages from a procedure may be useful in many cases:

- To manage mailing lists
- To send alert messages from scripts
- To send error messages to a program's development team
- ...

→

file: `simplemail.py` - function `send()` /1

```
import smtplib
import sys
import getpass

def send(mailhost, sender, recipients, subj, body, auth=None, debug=False):
    "Send a single message to a list of recipients"
    def log_debug(msg):
        "Print debug messages"
        if _debug:
            print("DBG>", msg)

    _debug = debug
    header = '\r\n'.join(("From: %s" % sender,
                          "To: %s" % ', '.join(recipients),
                          "Subject: %s" % subj,
                          ""))
```

file: simplemail.py - function send() /1

```
import smtplib ← 1
import sys
import getpass ← 2

def send(mailhost, sender, recipients, subj, body, auth=None, debug=False):
    "Send a single message to a list of recipients"  ← 3
    def log_debug(msg): ← 4
        "Print debug messages"
        if _debug:
            print("DBG>", msg)

    _debug = debug ← 5
    header = '\r\n'.join(("From: %s" % sender, ← 6
                          "To: %s" % ', '.join(recipients),
                          "Subject: %s" % subj,
                          ""))
```

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /1

```
import smtplib   <-- 1
import sys
import getpass   <-- 2

def send(mailhost, sender, recipients, subj, body, auth=None, debug=False):
    "Send a single message to a list of recipients"   <-- 3
    def log_debug(msg):   <-- 4
        "Print debug messages"
        if _debug:
            print("DBG>", msg)

    _debug = debug   <-- 5
    header = '\r\n'.join(("From: %s" % sender,   <-- 6
                          "To: %s" % ', '.join(recipients),
                          "Subject: %s" % subj,
                          ""))
```

1. The core module used in this example

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /1

```
import smtplib   <-- 1
import sys
import getpass   <-- 2

def send(mailhost, sender, recipients, subj, body, auth=None, debug=False):
    "Send a single message to a list of recipients"   <-- 3
    def log_debug(msg):   <-- 4
        "Print debug messages"
        if _debug:
            print("DBG>", msg)

    _debug = debug   <-- 5
    header = '\r\n'.join(("From: %s" % sender,   <-- 6
                          "To: %s" % ', '.join(recipients),
                          "Subject: %s" % subj,
                          ""))
```

1. The core module used in this example
2. This module is used only in the test section

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /1

```
import smtplib   <-- 1
import sys
import getpass   <-- 2

def send(mailhost, sender, recipients, subj, body, auth=None, debug=False):
    "Send a single message to a list of recipients"   <-- 3
    def log_debug(msg):   <-- 4
        "Print debug messages"
        if _debug:
            print("DBG>", msg)

    _debug = debug   <-- 5
    header = '\r\n'.join(("From: %s" % sender,   <-- 6
                          "To: %s" % ', '.join(recipients),
                          "Subject: %s" % subj,
                          ""))
```

1. The core module used in this example
2. This module is used only in the test section
3. `send()` is the function to be used to send e-mail messages

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /1

```
import smtplib          <-- 1
import sys
import getpass          <-- 2

def send(mailhost, sender, recipients, subj, body, auth=None, debug=False):
    "Send a single message to a list of recipients"          <-- 3
    def log_debug(msg):          <-- 4
        "Print debug messages"
        if _debug:
            print("DBG>", msg)

    _debug = debug          <-- 5
    header = '\r\n'.join(("From: %s" % sender,          <-- 6
                          "To: %s" % ', '.join(recipients),
                          "Subject: %s" % subj,
                          ""))
```

1. The core module used in this example
2. This module is used only in the test section
3. `send()` is the function to be used to send e-mail messages
4. Python allows to define a function within another function. As a result function `log_debug()` is ”visible” only within `send()`

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /1

```
import smtplib   <- 1
import sys
import getpass   <- 2

def send(mailhost, sender, recipients, subj, body, auth=None, debug=False):
    "Send a single message to a list of recipients"   <- 3
    def log_debug(msg):   <- 4
        "Print debug messages"
        if _debug:
            print("DBG>", msg)

    _debug = debug   <- 5
    header = '\r\n'.join(("From: %s" % sender,   <- 6
                          "To: %s" % ', '.join(recipients),
                          "Subject: %s" % subj,
                          ""))
```

1. The core module used in this example
2. This module is used only in the test section
3. `send()` is the function to be used to send e-mail messages
4. Python allows to define a function within another function. As a result function `log_debug()` is ”visible” only within `send()`
5. This enables/disables verbose output for debug

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /1

```
import smtplib ← 1
import sys
import getpass ← 2

def send(mailhost, sender, recipients, subj, body, auth=None, debug=False):
    "Send a single message to a list of recipients"  ← 3
    def log_debug(msg): ← 4
        "Print debug messages"
        if _debug:
            print("DBG>", msg)

    _debug = debug ← 5
    header = '\r\n'.join(("From: %s" % sender, ← 6
                          "To: %s" % ', '.join(recipients),
                          "Subject: %s" % subj,
                          ""))
```

1. The core module used in this example
2. This module is used only in the test section
3. `send()` is the function to be used to send e-mail messages
4. Python allows to define a function within another function. As a result function `log_debug()` is ”visible” only within `send()`
5. This enables/disables verbose output for debug
6. `header` is a string to format the mail message. It is generated by joining several lines of text

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /1

```
import smtplib   <- 1
import sys
import getpass   <- 2

def send(mailhost, sender, recipients, subj, body, auth=None, debug=False):
    "Send a single message to a list of recipients"   <- 3
    def log_debug(msg):   <- 4
        "Print debug messages"
        if _debug:
            print("DBG>", msg)

    _debug = debug   <- 5
    header = '\r\n'.join(("From: %s" % sender,   <- 6
                          "To: %s" % ', '.join(recipients),
                          "Subject: %s" % subj,
                          ""))
```

1. The core module used in this example
2. This module is used only in the test section
3. `send()` is the function to be used to send e-mail messages
4. Python allows to define a function within another function. As a result function `log_debug()` is ”visible” only within `send()`
5. This enables/disables verbose output for debug
6. `header` is a string to format the mail message. It is generated by joining several lines of text

→

file: `simplemail.py` - function `send()` /2

```
    body = '\r\n'.join(body)
    log_debug('Opening connection to: %s' % mailhost)
    try:
        server = smtplib.SMTP(mailhost)
        if auth:
            server.starttls()
            log_debug('Login as %s ...' % auth[0])
            server.login(auth[0], auth[1])
        log_debug('Sendmail ...')
        server.sendmail(sender, recipients, str(header)+str(body))
        log_debug('quit ...')
        server.quit()
    except Exception as excp:
        print("SMTP exception:", str(excp))
```

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /2

```
    body = '\r\n'.join(body)   <-- 1
    log_debug('Opening connection to: %s' % mailhost)
    try:
        server = smtplib.SMTP(mailhost)   <-- 2
        if auth:   <-- 3
            server.starttls()   <-- 3.1
            log_debug('Login as %s ...' % auth[0])
            server.login(auth[0], auth[1])   <-- 3.2
        log_debug('Sendmail ...')
        server.sendmail(sender, recipients, str(header)+str(body))   <-- 4
        log_debug('quit ...')
        server.quit()   <-- 5
    except Exception as excp:
        print("SMTP exception:", str(excp))
```

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /2

```
    body = '\r\n'.join(body)   <-- 1
    log_debug('Opening connection to: %s' % mailhost)
    try:
        server = smtplib.SMTP(mailhost)   <-- 2
        if auth:   <-- 3
            server.starttls()   <-- 3.1
            log_debug('Login as %s ...' % auth[0])
            server.login(auth[0], auth[1])   <-- 3.2
        log_debug('Sendmail ...')
        server.sendmail(sender, recipients, str(header)+str(body))   <-- 4
        log_debug('quit ...')
        server.quit()   <-- 5
    except Exception as excp:
        print("SMTP exception:", str(excp))
```

1. Here we make the mail body as a string

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /2

```
    body = '\r\n'.join(body)   <-- 1
    log_debug('Opening connection to: %s' % mailhost)
    try:
        server = smtplib.SMTP(mailhost)   <-- 2
        if auth:   <-- 3
            server.starttls()   <-- 3.1
            log_debug('Login as %s ...' % auth[0])
            server.login(auth[0], auth[1])   <-- 3.2
        log_debug('Sendmail ...')
        server.sendmail(sender, recipients, str(header)+str(body))   <-- 4
        log_debug('quit ...')
        server.quit()   <-- 5
    except Exception as excp:
        print("SMTP exception:", str(excp))
```

1. Here we make the mail body as a string
2. Activate the SMTP connection

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /2

```
    body = '\r\n'.join(body)   <-- 1
    log_debug('Opening connection to: %s' % mailhost)
    try:
        server = smtplib.SMTP(mailhost)   <-- 2
        if auth:   <-- 3
            server.starttls()   <-- 3.1
            log_debug('Login as %s ...' % auth[0])
            server.login(auth[0], auth[1])   <-- 3.2
        log_debug('Sendmail ...')
        server.sendmail(sender, recipients, str(header)+str(body))   <-- 4
        log_debug('quit ...')
        server.quit()   <-- 5
    except Exception as excp:
        print("SMTP exception:", str(excp))
```

1. Here we make the mail body as a string
2. Activate the SMTP connection
3. If authenticating ...

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /2

```
    body = '\r\n'.join(body)   <-- 1
    log_debug('Opening connection to: %s' % mailhost)
    try:
        server = smtplib.SMTP(mailhost)   <-- 2
        if auth:   <-- 3
            server.starttls()   <-- 3.1
            log_debug('Login as %s ...' % auth[0])
            server.login(auth[0], auth[1])   <-- 3.2
        log_debug('Sendmail ...')
        server.sendmail(sender, recipients, str(header)+str(body))   <-- 4
        log_debug('quit ...')
        server.quit()   <-- 5
    except Exception as excp:
        print("SMTP exception:", str(excp))
```

1. Here we make the mail body as a string
2. Activate the SMTP connection
3. If authenticating ...
   1. Initialize authentication protocol

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /2

```
    body = '\r\n'.join(body)   <-- 1
    log_debug('Opening connection to: %s' % mailhost)
    try:
        server = smtplib.SMTP(mailhost)   <-- 2
        if auth:   <-- 3
            server.starttls()   <-- 3.1
            log_debug('Login as %s ...' % auth[0])
            server.login(auth[0], auth[1])   <-- 3.2
        log_debug('Sendmail ...')
        server.sendmail(sender, recipients, str(header)+str(body))   <-- 4
        log_debug('quit ...')
        server.quit()   <-- 5
    except Exception as excp:
        print("SMTP exception:", str(excp))
```

1. Here we make the mail body as a string
2. Activate the SMTP connection
3. If authenticating ...
   1. Initialize authentication protocol
   2. Authenticate with userid and password

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /2

```
    body = '\r\n'.join(body)   <-- 1
    log_debug('Opening connection to: %s' % mailhost)
    try:
        server = smtplib.SMTP(mailhost)   <-- 2
        if auth:   <-- 3
            server.starttls()   <-- 3.1
            log_debug('Login as %s ...' % auth[0])
            server.login(auth[0], auth[1])   <-- 3.2
        log_debug('Sendmail ...')
        server.sendmail(sender, recipients, str(header)+str(body))   <-- 4
        log_debug('quit ...')
        server.quit()   <-- 5
    except Exception as excp:
        print("SMTP exception:", str(excp))
```

1. Here we make the mail body as a string
2. Activate the SMTP connection
3. If authenticating ...
   1. Initialize authentication protocol
   2. Authenticate with userid and password
4. Send the message

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /2

```
    body = '\r\n'.join(body)          <- 1
    log_debug('Opening connection to: %s' % mailhost)
    try:
        server = smtplib.SMTP(mailhost)          <- 2
        if auth:                     <- 3
            server.starttls()        <- 3.1
            log_debug('Login as %s ...' % auth[0])
            server.login(auth[0], auth[1])       <- 3.2
        log_debug('Sendmail ...')
        server.sendmail(sender, recipients, str(header)+str(body))   <- 4
        log_debug('quit ...')
        server.quit()                <- 5
    except Exception as excp:
        print("SMTP exception:", str(excp))
```

1. Here we make the mail body as a string
2. Activate the SMTP connection
3. If authenticating ...
   1. Initialize authentication protocol
   2. Authenticate with userid and password
4. Send the message
5. Close SMTP connection

Example 1: Sending e-mail

file: `simplemail.py` - function `send()` /2

```
    body = '\r\n'.join(body)   <-- 1
    log_debug('Opening connection to: %s' % mailhost)
    try:
        server = smtplib.SMTP(mailhost)   <-- 2
        if auth:   <-- 3
            server.starttls()   <-- 3.1
            log_debug('Login as %s ...' % auth[0])
            server.login(auth[0], auth[1])   <-- 3.2
        log_debug('Sendmail ...')
        server.sendmail(sender, recipients, str(header)+str(body))   <-- 4
        log_debug('quit ...')
        server.quit()   <-- 5
    except Exception as excp:
        print("SMTP exception:", str(excp))
```

1. Here we make the mail body as a string
2. Activate the SMTP connection
3. If authenticating ...
   1. Initialize authentication protocol
   2. Authenticate with userid and password
4. Send the message
5. Close SMTP connection

→

file: simplemail.py - Test code /1

```
def main():
    """
Test code usage:

    1. python simplemail.py server.net rec@pient.com
       (to send through unauthenticated server)

    2. python simplemail.py server.net userid rec@pient.com
       (to send through authenticated server)
"""
    sender = 'fake.sender@somewhere.eu'
    subj_fmt = 'Test message via %s'
    msg_body = ['', 'Test message generated by simplemail.py', '']

    if len(sys.argv) == 3: # Sending via non authenticating server
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[2]],
             subj, msg_body, debug=True)
```

file: simplemail.py - Test code /1

```
def main():
    """
Test code usage:

    1. python simplemail.py server.net rec@pient.com
       (to send through unauthenticated server)

    2. python simplemail.py server.net userid rec@pient.com
       (to send through authenticated server)
"""
    sender = 'fake.sender@somewhere.eu'
    subj_fmt = 'Test message via %s'
    msg_body = ['', 'Test message generated by simplemail.py', '']

    if len(sys.argv) == 3: # Sending via non authenticating server
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[2]],
             subj, msg_body, debug=True)
```

Example 1: Sending e-mail

file: `simplemail.py` - Test code /1

```
def main():
    """
Test code usage:

    1. python simplemail.py server.net rec@pient.com
       (to send through unauthenticated server)

    2. python simplemail.py server.net userid rec@pient.com
       (to send through authenticated server)
"""
    sender = 'fake.sender@somewhere.eu'
    subj_fmt = 'Test message via %s'
    msg_body = ['', 'Test message generated by simplemail.py', '']

    if len(sys.argv) == 3: # Sending via non authenticating server
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[2]],
             subj, msg_body, debug=True)
```

1 ← 1

2 ← 2

3,4 ← 3,4

5,6 ← 5,6

1. We set up test message elements

Example 1: Sending e-mail

file: `simplemail.py` - Test code /1

```
def main():
    """
Test code usage:

    1. python simplemail.py server.net rec@pient.com
       (to send through unauthenticated server)

    2. python simplemail.py server.net userid rec@pient.com
       (to send through authenticated server)
"""
    sender = 'fake.sender@somewhere.eu'
    subj_fmt = 'Test message via %s'
    msg_body = ['', 'Test message generated by simplemail.py', '']

    if len(sys.argv) == 3: # Sending via non authenticating server
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[2]],
             subj, msg_body, debug=True)
```

1 ← 1

2 ← 2

3,4 ← 3,4

5,6 ← 5,6

1. We set up test message elements
2. If the script is called with 2 arguments, send test message without authentication

Example 1: Sending e-mail

file: `simplemail.py` - Test code /1

```
def main():
    """
Test code usage:

    1. python simplemail.py server.net rec@pient.com
       (to send through unauthenticated server)

    2. python simplemail.py server.net userid rec@pient.com
       (to send through authenticated server)
"""
    sender = 'fake.sender@somewhere.eu'
    subj_fmt = 'Test message via %s'
    msg_body = ['', 'Test message generated by simplemail.py', '']

    if len(sys.argv) == 3: # Sending via non authenticating server
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[2]],
             subj, msg_body, debug=True)
```

1 ← 1

2 ← 2

3,4 ← 3,4

5,6 ← 5,6

1. We set up test message elements
2. If the script is called with 2 arguments, send test message without authentication
3. We call the sending function ...

Example 1: Sending e-mail

file: `simplemail.py` - Test code /1

```
def main():
    """
Test code usage:

    1. python simplemail.py server.net rec@pient.com
       (to send through unauthenticated server)

    2. python simplemail.py server.net userid rec@pient.com
       (to send through authenticated server)
"""
    sender = 'fake.sender@somewhere.eu'
    subj_fmt = 'Test message via %s'
    msg_body = ['', 'Test message generated by simplemail.py', '']

    if len(sys.argv) == 3: # Sending via non authenticating server
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[2]],
             subj, msg_body, debug=True)
```

1

2

3,4

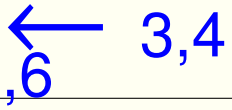
5,6

1. We set up test message elements
2. If the script is called with 2 arguments, send test message without authentication
3. We call the sending function ...
4. First argument is the mail server address, second argument is the recipient

Example 1: Sending e-mail

file: `simplemail.py` - Test code /1

```
def main():
    """
Test code usage:

    1. python simplemail.py server.net rec@pient.com
       (to send through unauthenticated server)

    2. python simplemail.py server.net userid rec@pient.com
       (to send through authenticated server)
"""
    sender = 'fake.sender@somewhere.eu'
    subj_fmt = 'Test message via %s'
    msg_body = ['', 'Test message generated by simplemail.py', '']

    if len(sys.argv) == 3: # Sending via non authenticating server
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[2]],
             subj, msg_body, debug=True)
```



1. We set up test message elements
2. If the script is called with 2 arguments, send test message without authentication
3. We call the sending function ...
4. First argument is the mail server address, second argument is the recipient
5. The optional argument `auth` is not specified: authentication is not performed

Example 1: Sending e-mail

file: `simplemail.py` - Test code /1

```
def main():
    """
Test code usage:

    1. python simplemail.py server.net rec@pient.com
       (to send through unauthenticated server)

    2. python simplemail.py server.net userid rec@pient.com
       (to send through authenticated server)
"""
    sender = 'fake.sender@somewhere.eu'
    subj_fmt = 'Test message via %s'
    msg_body = ['', 'Test message generated by simplemail.py', '']

    if len(sys.argv) == 3: # Sending via non authenticating server
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[2]],
             subj, msg_body, debug=True)
```

1 2 3,4 5,6

1. We set up test message elements
2. If the script is called with 2 arguments, send test message without authentication
3. We call the sending function ...
4. First argument is the mail server address, second argument is the recipient
5. The optional argument `auth` is not specified: authentication is not performed
6. The optional argument `debug` is set to `True` to have debugging printouts

Example 1: Sending e-mail

file: `simplemail.py` - Test code /1

```
def main():
    """
Test code usage:

    1. python simplemail.py server.net rec@pient.com
       (to send through unauthenticated server)

    2. python simplemail.py server.net userid rec@pient.com
       (to send through authenticated server)
"""
    sender = 'fake.sender@somewhere.eu'
    subj_fmt = 'Test message via %s'
    msg_body = ['', 'Test message generated by simplemail.py', '']

    if len(sys.argv) == 3: # Sending via non authenticating server
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[2]],
             subj, msg_body, debug=True)
```

1

2

3,4

5,6

1. We set up test message elements
2. If the script is called with 2 arguments, send test message without authentication
3. We call the sending function ...
4. First argument is the mail server address, second argument is the recipient
5. The optional argument `auth` is not specified: authentication is not performed
6. The optional argument `debug` is set to `True` to have debugging printouts

→

file: simplemail.py - Test code /2

```
    elif len(sys.argv) == 4: # Sending via authenticating server
        pwd = getpass.getpass(prompt="Password for %s: "%sys.argv[2])
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[3]],
             subj, msg_body, auth=(sys.argv[2], pwd), debug=True)
    else:
        print(main.__doc__)

if __name__ == '__main__':
    main()
```

file: `simplemail.py` - Test code /2

```
    elif len(sys.argv) == 4: # Sending via authenticating server
        pwd = getpass.getpass(prompt="Password for %s: "%sys.argv[2])
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[3]],
             subj, msg_body, auth=(sys.argv[2], pwd), debug=True)
    else:
        print(main.__doc__)

if __name__ == '__main__':
    main()
```

Example 1: Sending e-mail

file: `simplemail.py` - Test code /2

```
    elif len(sys.argv) == 4: # Sending via authenticating server
        pwd = getpass.getpass(prompt="Password for %s: "%sys.argv[2])
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[3]],
             subj, msg_body, auth=(sys.argv[2], pwd), debug=True)
    else:
        print(main.__doc__)

if __name__ == '__main__':
    main()
```

1 ← 1
2 ← 2
3 ← 3
4 ← 4
5 ← 5
6 ← 6

1. With three arguments, send via an authenticating server

Example 1: Sending e-mail

file: `simplemail.py` - Test code /2

```
    elif len(sys.argv) == 4: # Sending via authenticating server   <- 1
        pwd = getpass.getpass(prompt="Password for %s: "%sys.argv[2])   <- 2
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[3]],   <- 3
             subj, msg_body, auth=(sys.argv[2], pwd), debug=True)   <- 4
    else:   <- 5
        print(main.__doc__)

if __name__ == '__main__':   <- 6
    main()
```

1. With three arguments, send via an authenticating server
2. Prompt for password

Example 1: Sending e-mail

file: `simplemail.py` - Test code /2

```
    elif len(sys.argv) == 4: # Sending via authenticating server
        pwd = getpass.getpass(prompt="Password for %s: "%sys.argv[2])
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[3]],
             subj, msg_body, auth=(sys.argv[2], pwd), debug=True)
    else:
        print(main.__doc__)

if __name__ == '__main__':
    main()
```

1. With three arguments, send via an authenticating server
2. Prompt for password
3. Argument 1 is the mail server address, argument 2 is the access userid, argument 3 is the recipient

Example 1: Sending e-mail

file: `simplemail.py` - Test code /2

```
    elif len(sys.argv) == 4: # Sending via authenticating server   <- 1
        pwd = getpass.getpass(prompt="Password for %s: "%sys.argv[2])   <- 2
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[3]],   <- 3
             subj, msg_body, auth=(sys.argv[2], pwd), debug=True)   <- 4
    else:   <- 5
        print(main.__doc__)

if __name__ == '__main__':   <- 6
    main()
```

1. With three arguments, send via an authenticating server
2. Prompt for password
3. Argument 1 is the mail server address, argument 2 is the access userid, argument 3 is the recipient
4. When optional argument `auth` is specified it contains the authorization data (userid and password)

Example 1: Sending e-mail

file: `simplemail.py` - Test code /2

```
    elif len(sys.argv) == 4: # Sending via authenticating server
        pwd = getpass.getpass(prompt="Password for %s: "%sys.argv[2])
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[3]],
             subj, msg_body, auth=(sys.argv[2], pwd), debug=True)
    else:
        print(main.__doc__)

if __name__ == '__main__':
    main()
```

1. With three arguments, send via an authenticating server
2. Prompt for password
3. Argument 1 is the mail server address, argument 2 is the access userid, argument 3 is the recipient
4. When optional argument `auth` is specified it contains the authorization data (userid and password)
5. If an unexpected number of arguments is specified, just provide some help

Example 1: Sending e-mail

file: `simplemail.py` - Test code /2

```
    elif len(sys.argv) == 4: # Sending via authenticating server   <- 1
        pwd = getpass.getpass(prompt="Password for %s: "%sys.argv[2])   <- 2
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[3]],   <- 3
             subj, msg_body, auth=(sys.argv[2], pwd), debug=True)   <- 4
    else:   <- 5
        print(main.__doc__)

if __name__ == '__main__':   <- 6
    main()
```

1. With three arguments, send via an authenticating server
2. Prompt for password
3. Argument 1 is the mail server address, argument 2 is the access userid, argument 3 is the recipient
4. When optional argument `auth` is specified it contains the authorization data (userid and password)
5. If an unexpected number of arguments is specified, just provide some help
6. Conditional execution of the test function `main()`

Example 1: Sending e-mail

file: `simplemail.py` - Test code /2

```
    elif len(sys.argv) == 4: # Sending via authenticating server   <- 1
        pwd = getpass.getpass(prompt="Password for %s: "%sys.argv[2])   <- 2
        subj = subj_fmt % sys.argv[1]
        print("Sending: ", subj)
        send(sys.argv[1], sender, [sys.argv[3]],   <- 3
             subj, msg_body, auth=(sys.argv[2], pwd), debug=True)   <- 4
    else:   <- 5
        print(main.__doc__)

if __name__ == '__main__':   <- 6
    main()
```

1. With three arguments, send via an authenticating server
2. Prompt for password
3. Argument 1 is the mail server address, argument 2 is the access userid, argument 3 is the recipient
4. When optional argument `auth` is specified it contains the authorization data (userid and password)
5. If an unexpected number of arguments is specified, just provide some help
6. Conditional execution of the test function `main()`

→

The End

End of Part II