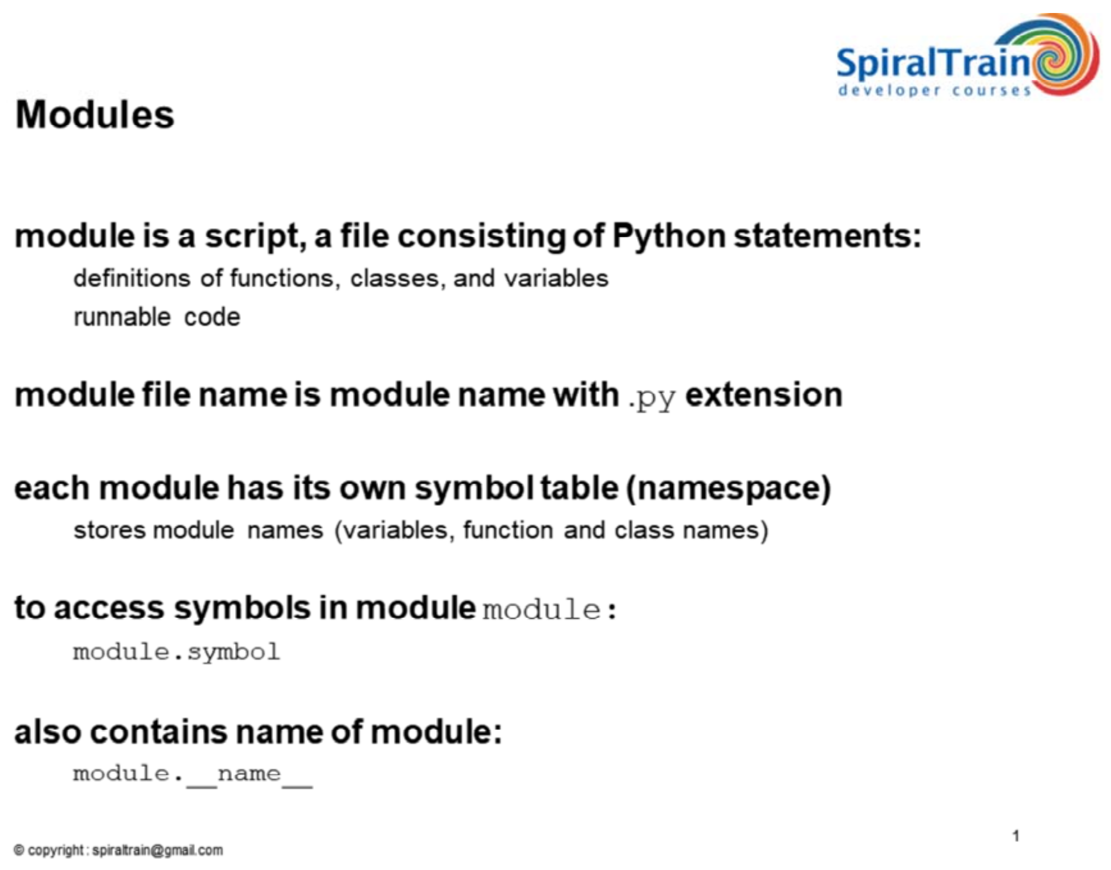


Modules

module is a script, a file consisting of Python statements:

- definitions of functions, classes, and variables
- runnable code

module file name is module name with `.py` extension

each module has its own symbol table (namespace)

- stores module names (variables, function and class names)

to access symbols in module `module`:

- `module.symbol`

also contains name of module:

- `module.__name__`

© copyright : spiraltrain@gmail.com

1

You have seen how you can reuse code in your program by defining functions once. What if you wanted to reuse a number of functions in other programs that you write? As you might have guessed, the answer is modules.

There are various methods of writing modules, but the simplest way is to create a file with a .py extension that contains functions and variables.

Another method is to write the modules in the native language in which the Python interpreter itself was written. For example, you can write modules in the C programming language and when compiled, they can be used from your Python code when using the standard Python interpreter.

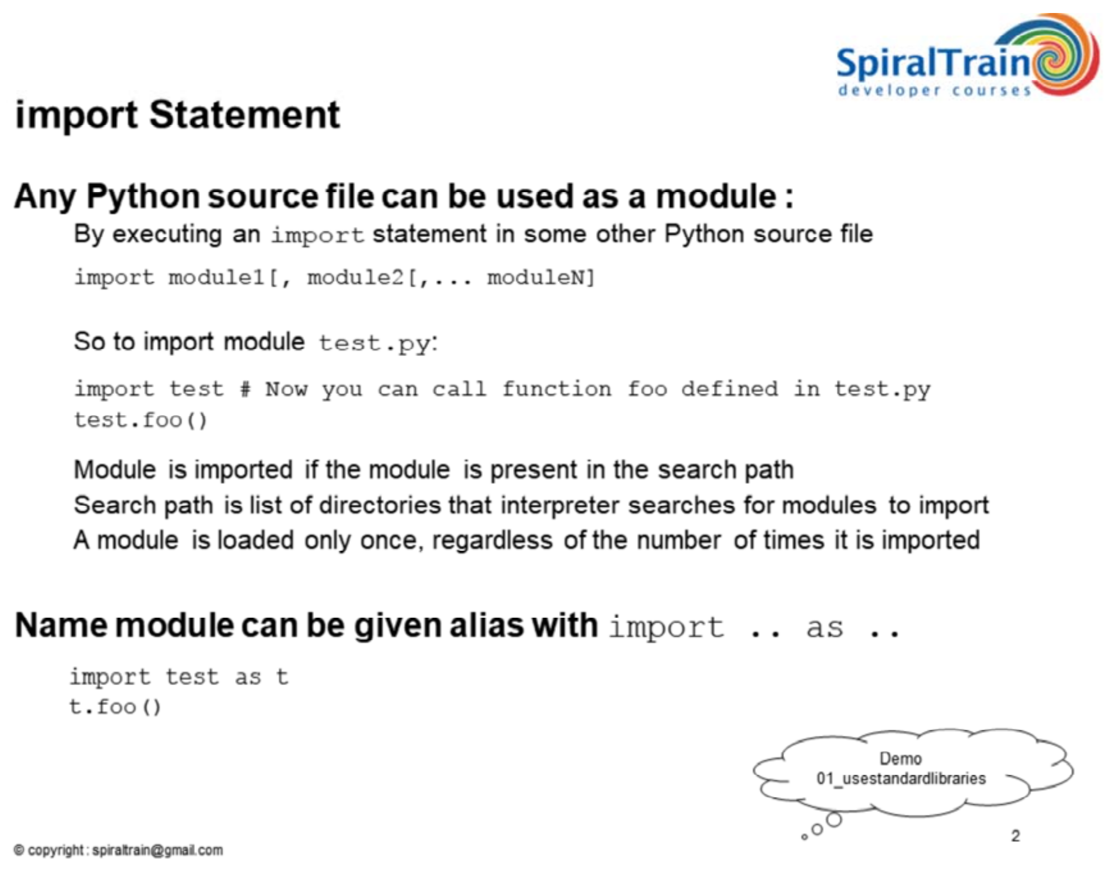
A module can be *imported* by another program to make use of its functionality. This is how we can use the Python standard library as well. First, we will see how to use the standard library modules.

```
import sys
print('The command line arguments are:')
for i in sys.argv:
    print(i)
print('\n\nThe PYTHONPATH is', sys.path, '\n')
```

First, we *import* the sys module using the import statement. Basically, this translates to us telling Python that we want to use this module. The sys module contains functionality related to the Python interpreter and its environment i.e. the *sys*tem.

When Python executes the import sys statement, it looks for the sys module. In this case, it is one of the built-in modules, and hence Python knows where to find it.

If it was not a compiled module i.e. a module written in Python, then the Python interpreter will search for it in the directories listed in its sys.path variable. If the module is found, then the statements in the body of that module are run and the module is made *available* for you to use. Note that the initialization is done only the *first* time that we import a module.



import Statement

Any Python source file can be used as a module :

By executing an `import` statement in some other Python source file

```
import module1[, module2[,... moduleN]
```

So to import module `test.py`:

```
import test # Now you can call function foo defined in test.py
test.foo()
```

Module is imported if the module is present in the search path
Search path is list of directories that interpreter searches for modules to import
A module is loaded only once, regardless of the number of times it is imported

Name module can be given alias with `import .. as ..`

```
import test as t
t.foo()
```

Demo
01_usestandardlibraries

© copyright : spiraltrain@gmail.com

2

The argv variable in the sys module is accessed using the dotted notation i.e. sys.argv. It clearly indicates that this name is part of the sys module. Another advantage of this approach is that the name does not clash with any argv variable used in your program.

The sys.argv variable is a *list* of strings. Specifically, the sys.argv contains the list of *command line arguments* i.e. the arguments passed to your program using the command line.

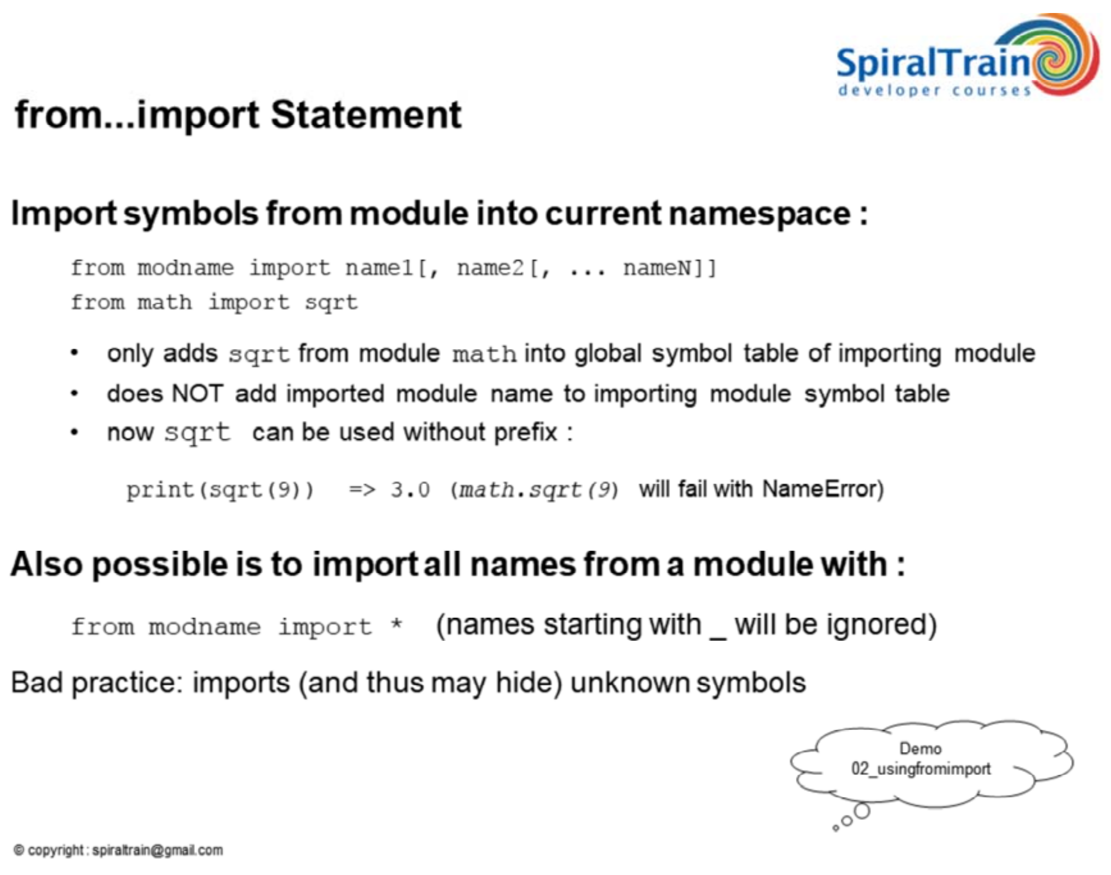
If you are using an IDE to write and run these programs, look for a way to specify command line arguments to the program in the menus.

Here, when we execute python using_sys.py we are arguments, we run the module using_sys.py with the python command and the other things that follow are arguments passed to the program. Python stores the command line arguments in the sys.argv variable for us to use.

Remember, the name of the script running is always the first argument in the sys.argv list. So, in this case we will have 'using_sys.py' as sys.argv[0], 'we' as sys.argv[1], 'are' as sys.argv[2] and 'arguments' as sys.argv[3]. Notice that Python starts counting from 0 and not 1.

The sys.path contains the list of directory names where modules are imported from. Observe that the first string in sys.path is empty - this empty string indicates that the current directory is also part of the sys.path which is same as the PYTHONPATH environment variable. This means that you can directly import modules located in the current directory. Otherwise, you will have to place your module in one of the directories listed in sys.path.

Note that the current directory is the directory from which the program is launched. Run import os; print(os.getcwd()) to find out the current directory of your program.



from...import Statement

Import symbols from module into current namespace :

```
from modname import name1[, name2[, ... nameN]]
from math import sqrt
```

- only adds `sqrt` from module `math` into global symbol table of importing module
- does NOT add imported module name to importing module symbol table
- now `sqrt` can be used without prefix :

  `print(sqrt(9))   => 3.0  (math.sqrt(9)` will fail with NameError)

Also possible is to import all names from a module with :

`from modname import *` (names starting with _ will be ignored)

Bad practice: imports (and thus may hide) unknown symbols

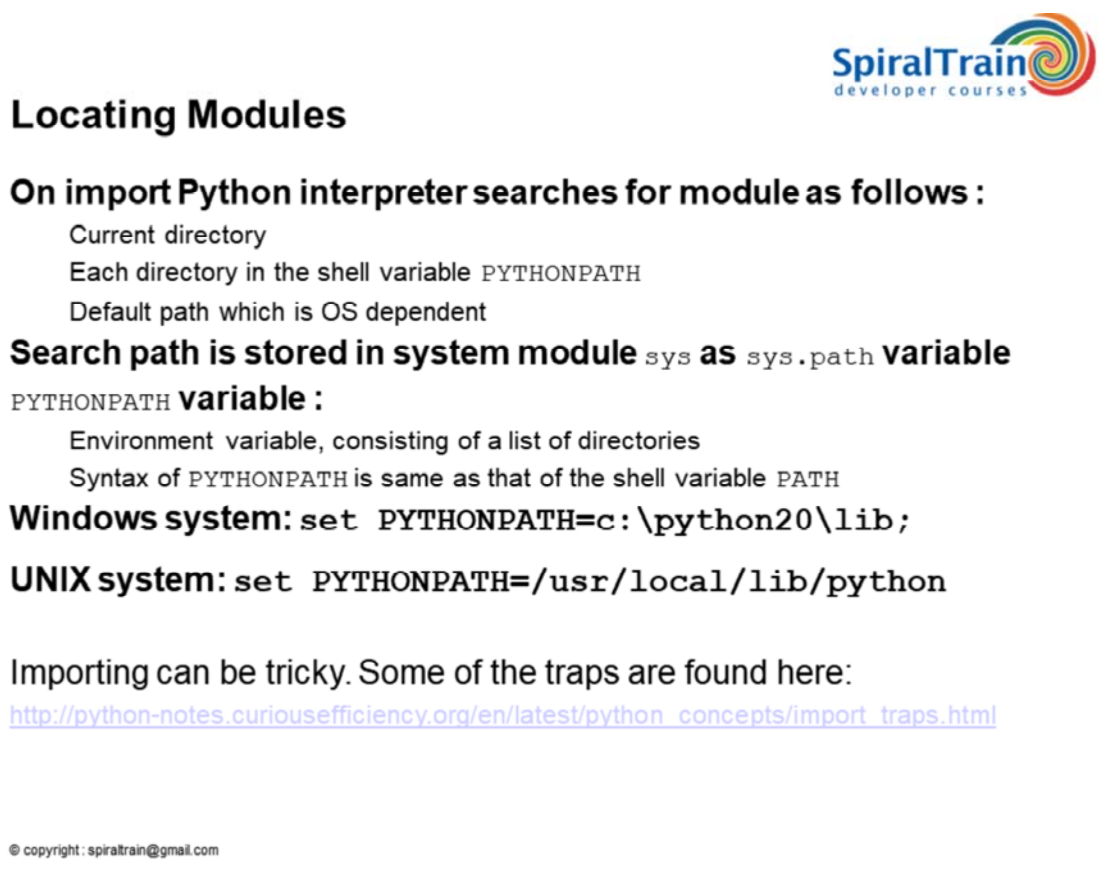
Demo
02_usingfromimport

© copyright : spiraltrain@gmail.com

If you want to directly import the argv variable into your program (to avoid typing the sys. everytime for it), then you can use the from sys import argv statement. If you want to import all the names used in the sys module, then you can use the from sys import * statement. This works for any module. In general, you should avoid using this statement and use the import statement instead since your program will avoid name clashes and will be more readable. Example:-

```
from math import *
n = int(input("Enter range:-  "))
p = [2, 3]
count = 2
a = 5
while(count < n):
    b = 0
    for i in range(2, a):
     if(i <= sqrt(a)):
          if(a % i == 0):
          #print a, "is not a prime"
          b = 1
          else:
          pass
     if(b != 1):
         #print a, "is a prime"
         p = p + [a]
```

```
          count = count + 1
      a = a + 2
print p
```



Locating Modules

On import Python interpreter searches for module as follows :

- Current directory
- Each directory in the shell variable `PYTHONPATH`
- Default path which is OS dependent

Search path is stored in system module `sys` as `sys.path` variable

`PYTHONPATH` variable :

- Environment variable, consisting of a list of directories
- Syntax of `PYTHONPATH` is same as that of the shell variable `PATH`

Windows system: `set PYTHONPATH=c:\python20\lib;`

UNIX system: `set PYTHONPATH=/usr/local/lib/python`

Importing can be tricky. Some of the traps are found here:
http://python-notes.curiousefficiency.org/en/latest/python_concepts/import_traps.html

© copyright : spiraltrain@gmail.com

For modules to be available for use, the Python interpreter must be able to locate the module file. Python has a set of directories in which it looks for module files. This set of directories is called the search path, and is analogous to the PATH environment variable used by an operating system to locate an executable file.

Python's search path is built from a number of sources:

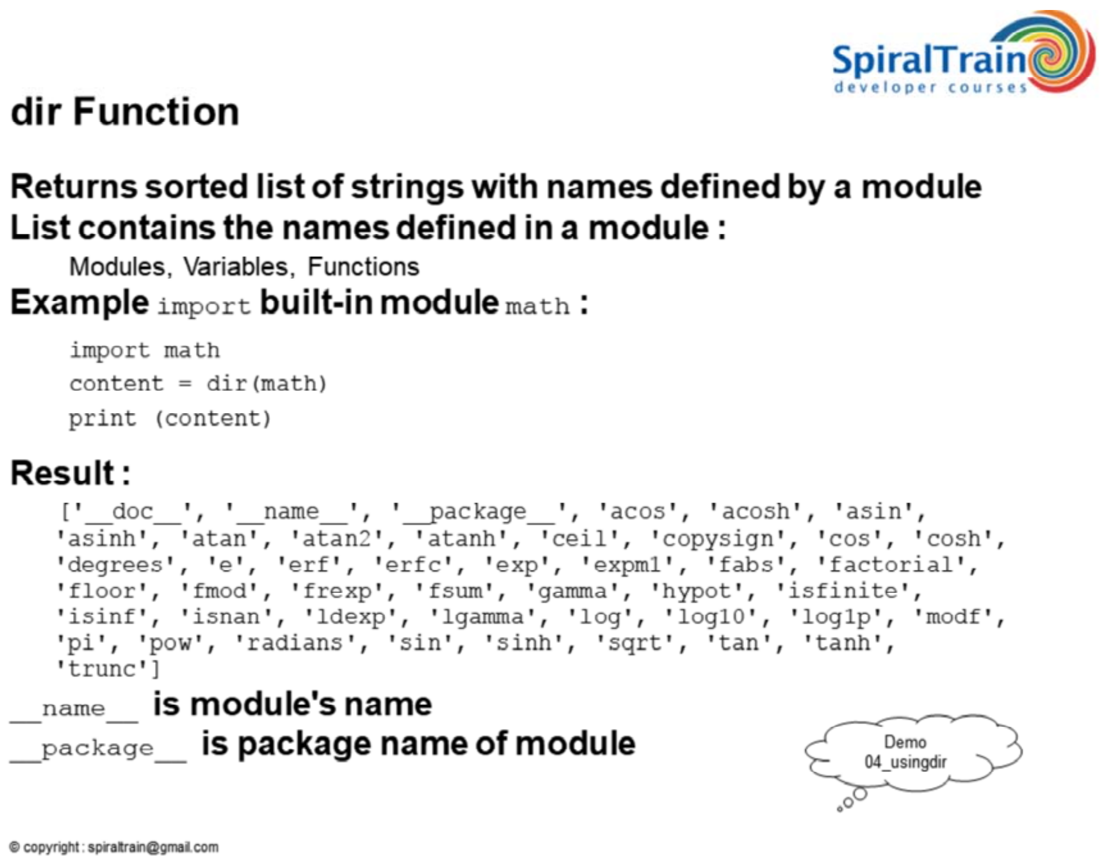
PYTHONHOME is used to define directories that are part of the Python installation. If this environment variable is not defined, then a standard directory structure is used. For Windows, the standard location is based on the directory into which Python is installed. For most Linux environments, Python is installed under /usr/local, and the libraries can be found there. For Mac OS, the home directory is under /Library/Frameworks/Python.framework.

PYTHONPATH is used to add directories to the path. This environment variable is formatted like the OS PATH variable, with a series of filenames separated by :'s (or ;'s for Windows).

Script Directory. If you run a Python script, that script's directory is placed first on the search path so that locally-defined moules will be used instead of built-in modules of the same name.

The site module's locations are also added. (This can be disabled by starting Python with the -S option.) The site module will use the PYTHONHOME location(s) to create up to four additional directories. Generally, the most interesting one is the site-packages directory. This directory is a handy place to put additional modules you've downloaded. Additionally, this directory can contain .PTH files. The site module reads .PTH files and puts the named directories onto the search path.

The search path is defined by the path variable in the sys module. If we import sys, we can display sys.path. This is very handy for debugging. When debugging shell scripts, it can help to run 'python -c 'import sys; print sys.path' just to see parts of the Python environment settings.



dir Function

Returns sorted list of strings with names defined by a module
List contains the names defined in a module :

Modules, Variables, Functions

Example `import` built-in module `math` :

```
import math
content = dir(math)
print (content)
```

Result :

```
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite',
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf',
'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'trunc']
```

`__name__` **is module's name**
`__package__` **is package name of module**

Demo
04_usingdir

© copyright : spiraltrain@gmail.com

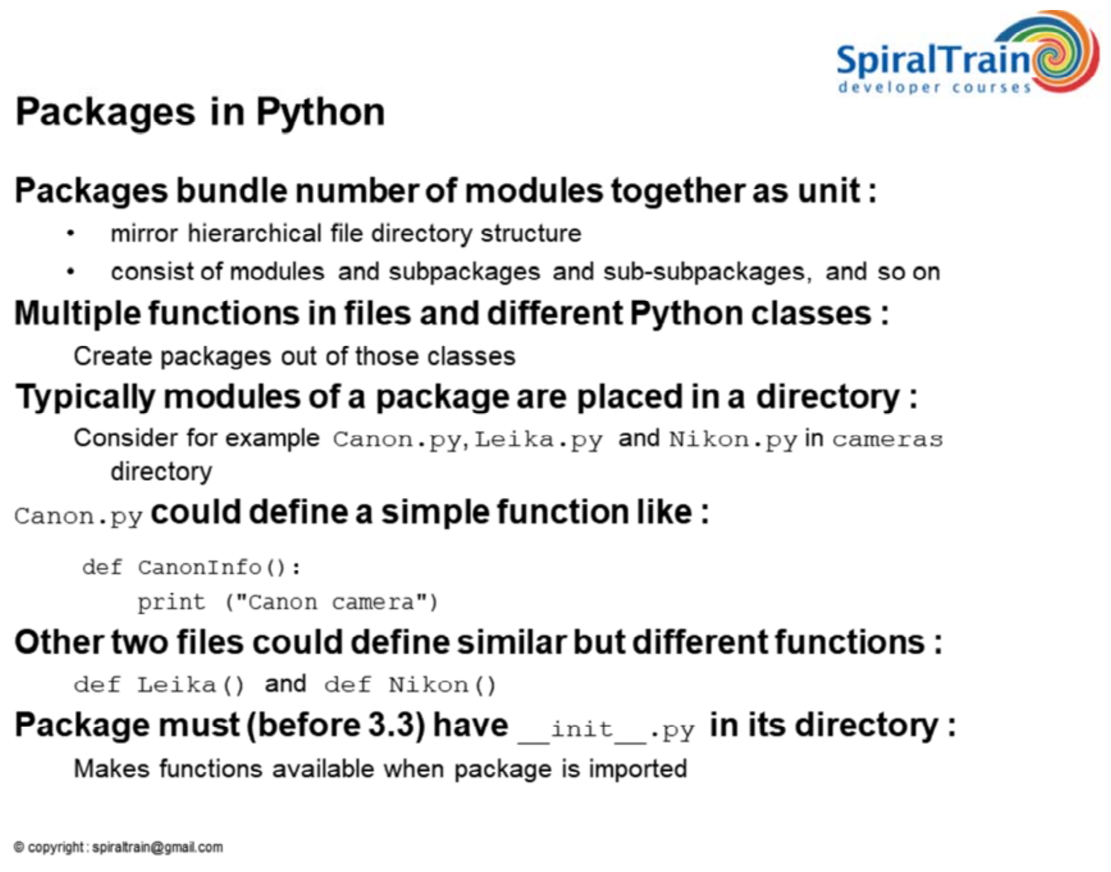
You can use the built-in `dir` function to list the identifiers that an object defines. For example, for a module, the identifiers include the functions, classes and variables defined in that module.

When you supply a module name to the dir() function, it returns the list of the names defined in that module. When no argument is applied to it, it returns the list of names defined in the current module.

```
$ python
>>> import sys # get list of attributes, in this case, for
the sys module
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__',
tderr__', '__stdin__', '__stdout__', '_clear_type_cache',
'_current_frames', '_getframe', 'api_version', 'argv',
', 'meta_path', 'modules', 'path', 'path_hooks',
m', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setprofile',
', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',
fo', 'warnoptions', 'winver']
>>> dir() # get list of attributes for current module
['__builtins__', '__doc__', '__name__', '__package__', 'sys‘]
```

First, we see the usage of dir on the imported sys module. We can see the huge list of attributes that it contains.

Next, we use the dir function without passing parameters to it. By default, it returns the list of attributes for the current module. Notice that the list of imported modules is also part of this list.



Packages in Python

Packages bundle number of modules together as unit :

- mirror hierarchical file directory structure
- consist of modules and subpackages and sub-subpackages, and so on

Multiple functions in files and different Python classes :

Create packages out of those classes

Typically modules of a package are placed in a directory :

Consider for example `Canon.py`, `Leika.py` and `Nikon.py` in `cameras` directory

`Canon.py` **could define a simple function like :**

```
def CanonInfo():
    print ("Canon camera")
```

Other two files could define similar but different functions :

`def Leika()` and `def Nikon()`

Package must (before 3.3) have `__init__.py` **in its directory :**

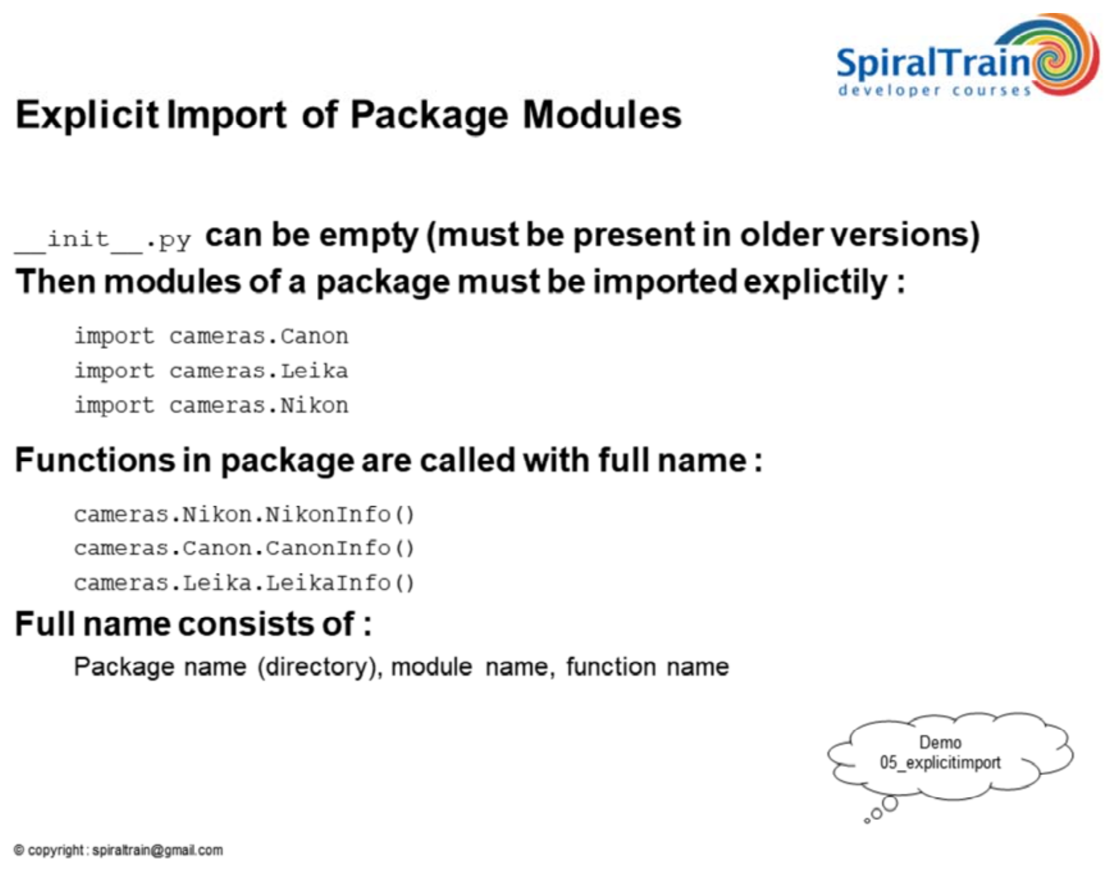
Makes functions available when package is imported

© copyright : spiraltrain@gmail.com

A package is a collection of Python modules. Packages allow us to structure a collection of modules. A package is a directory that contains modules. Having a directory of modules allows us to have modules contained within other modules. This allows us to use qualified module names, clarifying the organization of our software. We can, for example, have several simulations of casino games. Rather than pile all of our various files into a single, flat directory, we might have the following kind of directory structure. (This isn't technically complete, it needs a few additional files.)

```
casino/
    craps/
        dice.py
        game.py
        player.py
    roulette/
        wheel.py
        game.py
        player.py
    blackjack/
        cards.py
        game.py
        player.py
```

```
strategy/
    basic.py
    martingale.py
    bet1326.py
    cancellation.py
```



Explicit Import of Package Modules

`__init__.py` **can be empty (must be present in older versions)**
Then modules of a package must be imported explicitly :

```
import cameras.Canon
import cameras.Leika
import cameras.Nikon
```

Functions in package are called with full name :

```
cameras.Nikon.NikonInfo()
cameras.Canon.CanonInfo()
cameras.Leika.LeikaInfo()
```

Full name consists of :

Package name (directory), module name, function name

Demo
05_explicitimport

© copyright : spiraltrain@gmail.com

Given this directory structure, our overall simulation might include statements like the following.

```
import craps.game, craps.player
import strategy.basic as betting
class MyPlayer( craps.player.Player ):
    def __init__( self, stake, turns ):
        betting.initialize(self)
```

We imported the game and player modules from the craps package. We imported the basic module from the strategy package. We defined a new player based on a class named Player in the craps.player package.

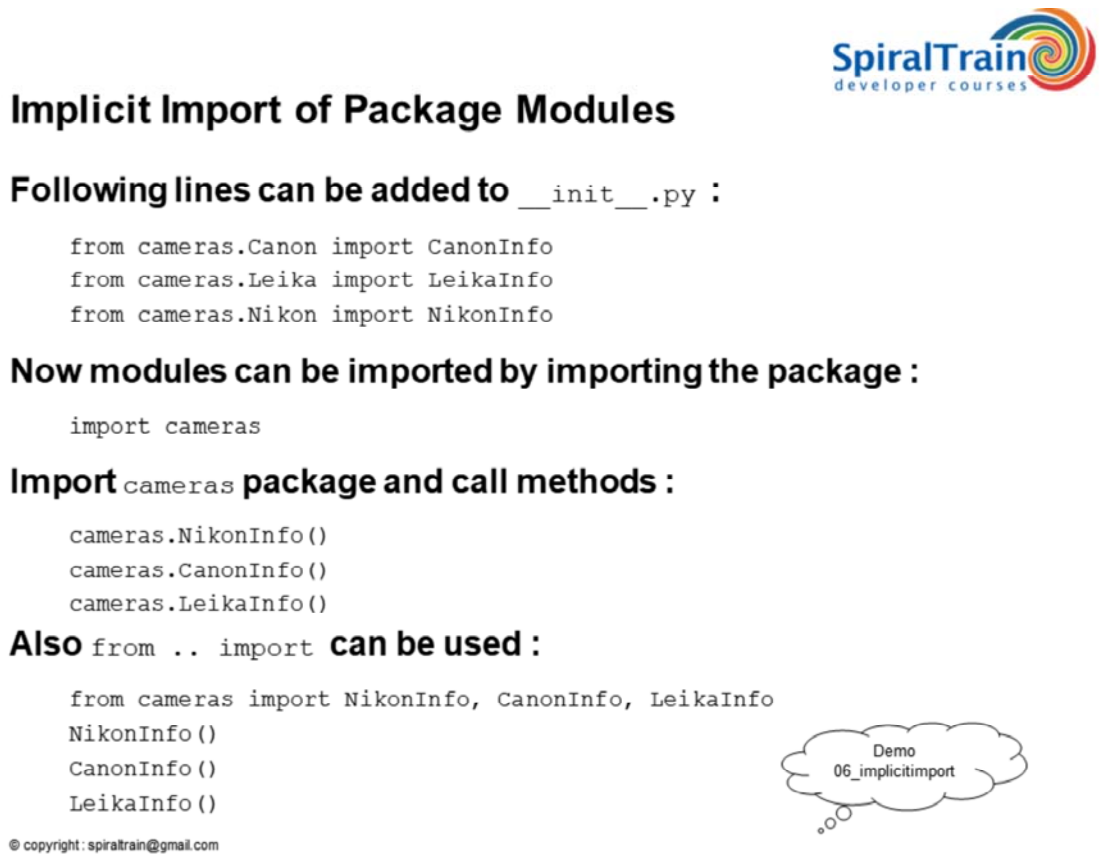
We have a number of alternative betting strategies, all collected under the strategy package. When we import a particular betting strategy, we name the module betting. We can then change to a different betting strategy by changing the **import** statement.

There are two reasons for using a package of modules.

There are a lot of modules, and the package structure clarifies the relationships among the modules. If we have several modules related to the game of craps, we might have the urge to create a craps_game.py module and a craps_player.py module. As soon as we start structuring the module names to show a relationship, we can use a package instead.

There are alternative implementations, and the package contains polymorphic modules. In this case, we will often use an import *package.alternative* as *interface* kind of **import** statement. This is often used for interfaces and drivers to isolate the interface details and provide a uniform API to the rest of the Python application.

It is possible to go overboard in package structuring. The general rule is to keep the package structure relatively flat. Having only one module at the bottom of deeply-nested packages isn't really very informative or helpful.



Implicit Import of Package Modules

Following lines can be added to `__init__.py` :

```
from cameras.Canon import CanonInfo
from cameras.Leika import LeikaInfo
from cameras.Nikon import NikonInfo
```

Now modules can be imported by importing the package :

```
import cameras
```

Import `cameras` package and call methods :

```
cameras.NikonInfo()
cameras.CanonInfo()
cameras.LeikaInfo()
```

Also `from .. import` can be used :

```
from cameras import NikonInfo, CanonInfo, LeikaInfo
NikonInfo()
CanonInfo()
LeikaInfo()
```

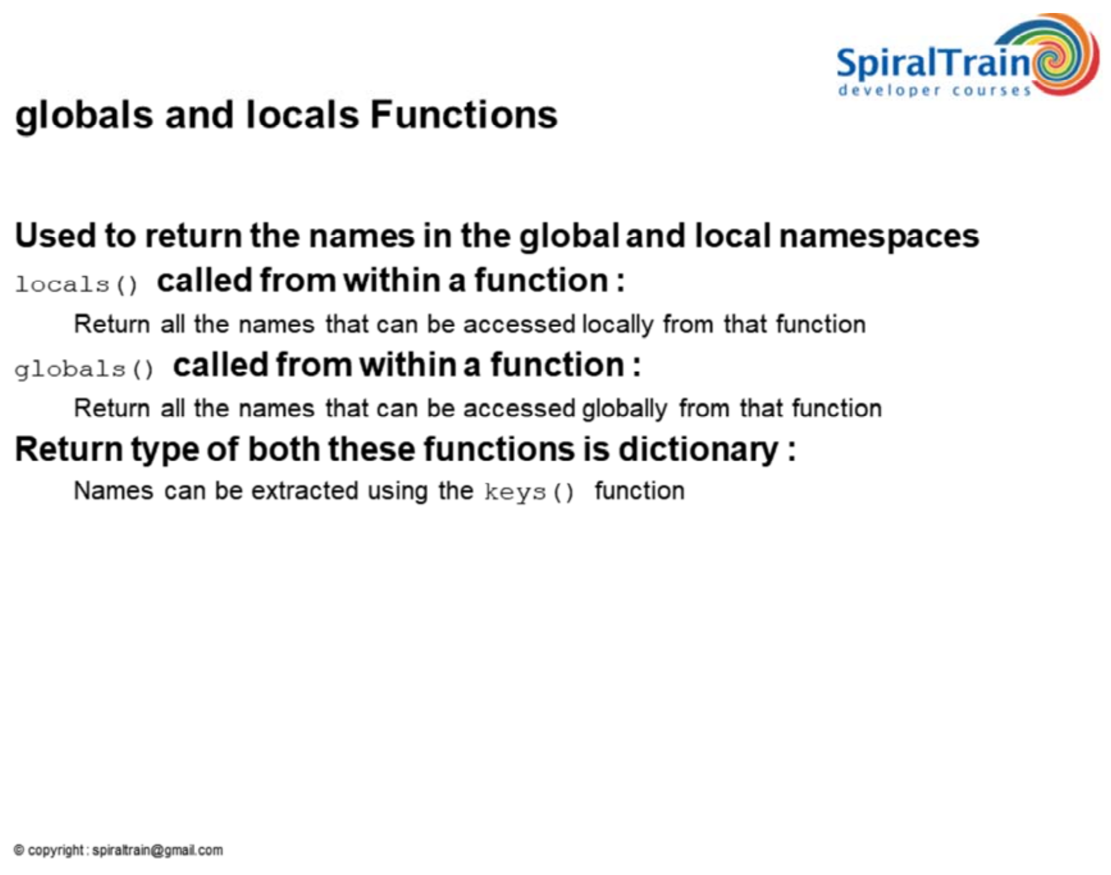
© copyright : spiraltrain@gmail.com

__init__. Valid Python names are composed of letters, digits and underscores. See the section called “Variables” for more informaiton.
The __init__ module is often an empty file, __init__.py in the package directory. Nothing else is required to make a directory into a package. The __init__.py file, however, is essential. Without it, you'll get an ImportError.
For example, consider a number of modules related to the definition of cards. We might have the following files.

```
cards/
    __init__.py
    standard.py
    blackjack.py
    poker.py
```

The cards.standard module would provide the base definition of card as an object with suit and rank. The cards.blackjack module would provide the subclasses of cards that we looked at in the section called “Blackjack Hands”. The cards.poker module would provided the subclasses of cards that we looked at in the section called “Poker Hands”.
The cards.blackjack module and the cards.poker module should both import the cards.standard module to get the base definition for the Card and Deck classes.
The __init__ module. The __init__ module is the "initialization" module in a package. It is processed the first time that a package name is encountered in an import statement. In effect, it initializes Python's understanding of the package. Since it is always loaded, it is also effectively the default module in a package. There are a number of consequences to this.
We can import the package, without naming a specific module. In this case we've imported just the initialization module, __init__. If this is part of our design, we'll put some kind of default or top-level definitions in the __init__ module.

We can import a specific module from the package. In this case, we also import the initialization module along with the requested module. In this case, the __init__ module can provide additional definitions; or it can simply be empty.



globals and locals Functions

Used to return the names in the global and local namespaces

`locals()` **called from within a function :**

Return all the names that can be accessed locally from that function

`globals()` **called from within a function :**

Return all the names that can be accessed globally from that function

Return type of both these functions is dictionary :

Names can be extracted using the `keys()` function

© copyright : spiraltrain@gmail.com

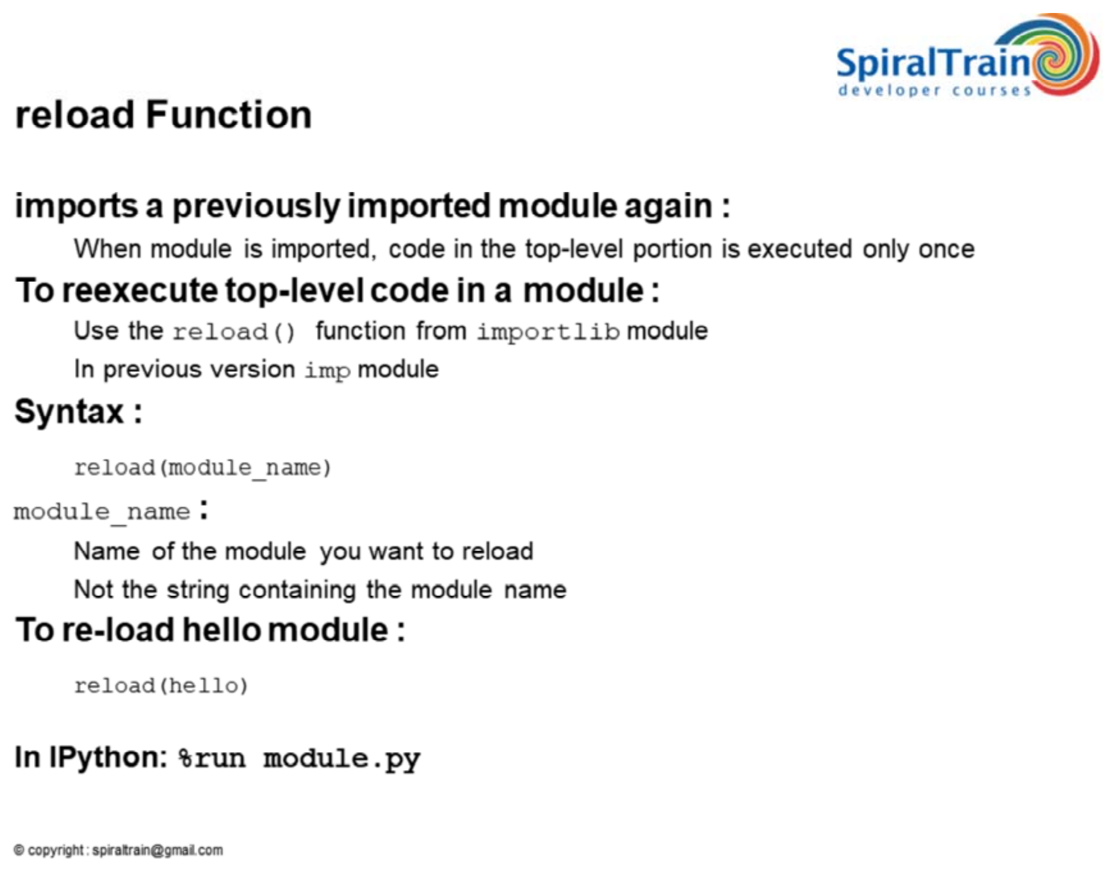
In our cards example, above, we would do well to make the __init__ module define the basic Card and Deck classes. If we import the package, cards, we get the default module, __init__, which gives us cards.Card and cards.Deck. If we import a specific module like cards.blackjack, we get the __init__ module plus the named module within the package.

Package Use

If the __init__ module in a package is empty, the package is little more than a collection of module files. In this case, we don't generally make direct use of a package. We merely mention it in an import statement: import cards.poker.

On other hand, if the __init__ module has some definitions, we can import the package itself. Importing a package just imports the __init__ module from the package directory. In this case, we mention the package in an import statement: import cards.

Even if the __init__ module has some definitions in it, we can always import a specific module from within the package. Indeed, it is possible for the __init__ module in a package is to do things like adjust the search path prior to locating individual module files.



reload Function

imports a previously imported module again :

When module is imported, code in the top-level portion is executed only once

To reexecute top-level code in a module :

Use the `reload()` function from `importlib` module

In previous version `imp` module

Syntax :

```
reload(module_name)
```

`module_name` :

Name of the module you want to reload

Not the string containing the module name

To re-load hello module :

```
reload(hello)
```

In IPython: `%run module.py`

© copyright : spiraltrain@gmail.com

Reload a previously imported module. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the module argument).

When reload(module) is executed:

Python modules’ code is recompiled and the module-level code reexecuted, defining a new set of objects which are bound to names in the module’s dictionary. The init function of extension modules is not called a second time.

As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.

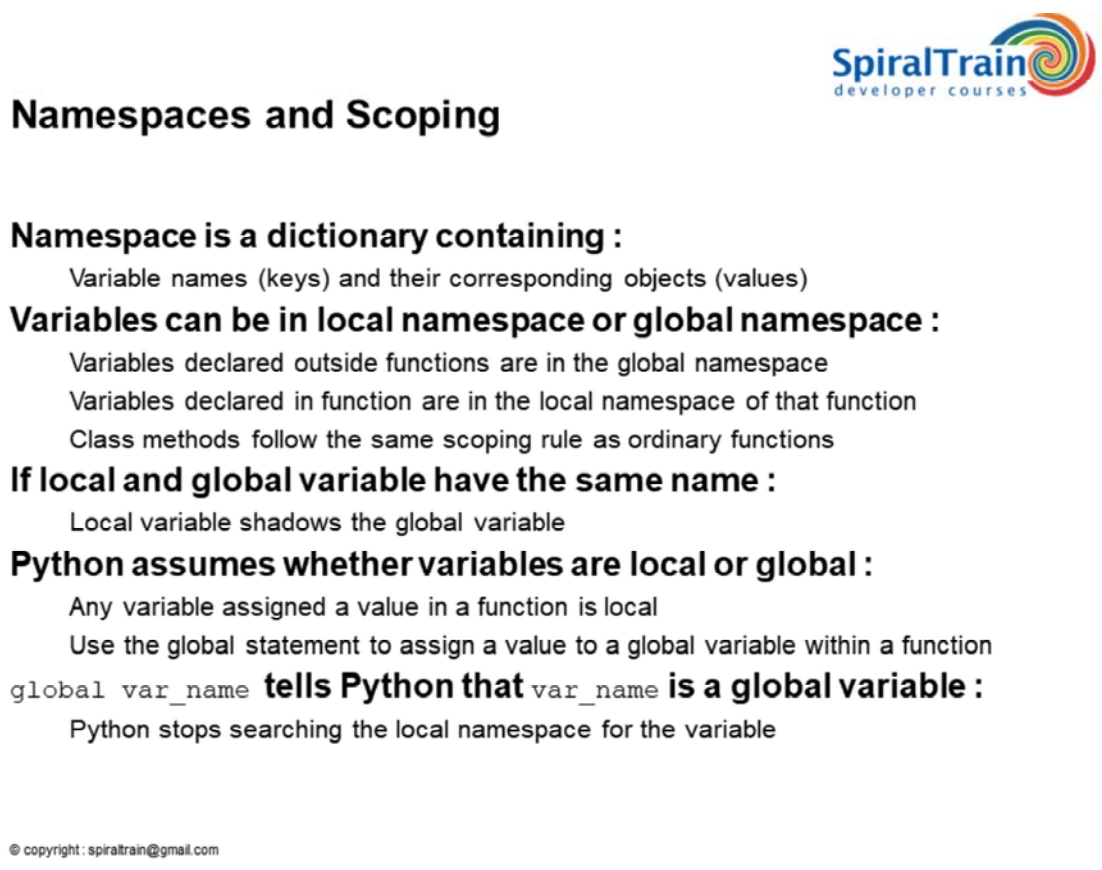
The names in the module namespace are updated to point to any new or changed objects.

Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

If a module is syntactically correct but its initialization fails, the first import statement for it does not bind its name locally, but does store a (partially initialized) module object in sys.modules. To reload the module you must first import it again (this will bind the name to the partially initialized module object) before you can reload() it.

When a module is reloaded, its dictionary (containing the module’s global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains.



Namespaces and Scoping

Namespace is a dictionary containing :

Variable names (keys) and their corresponding objects (values)

Variables can be in local namespace or global namespace :

Variables declared outside functions are in the global namespace

Variables declared in function are in the local namespace of that function

Class methods follow the same scoping rule as ordinary functions

If local and global variable have the same name :

Local variable shadows the global variable

Python assumes whether variables are local or global :

Any variable assigned a value in a function is local

Use the global statement to assign a value to a global variable within a function

`global var_name` **tells Python that** `var_name` **is a global variable :**

Python stops searching the local namespace for the variable

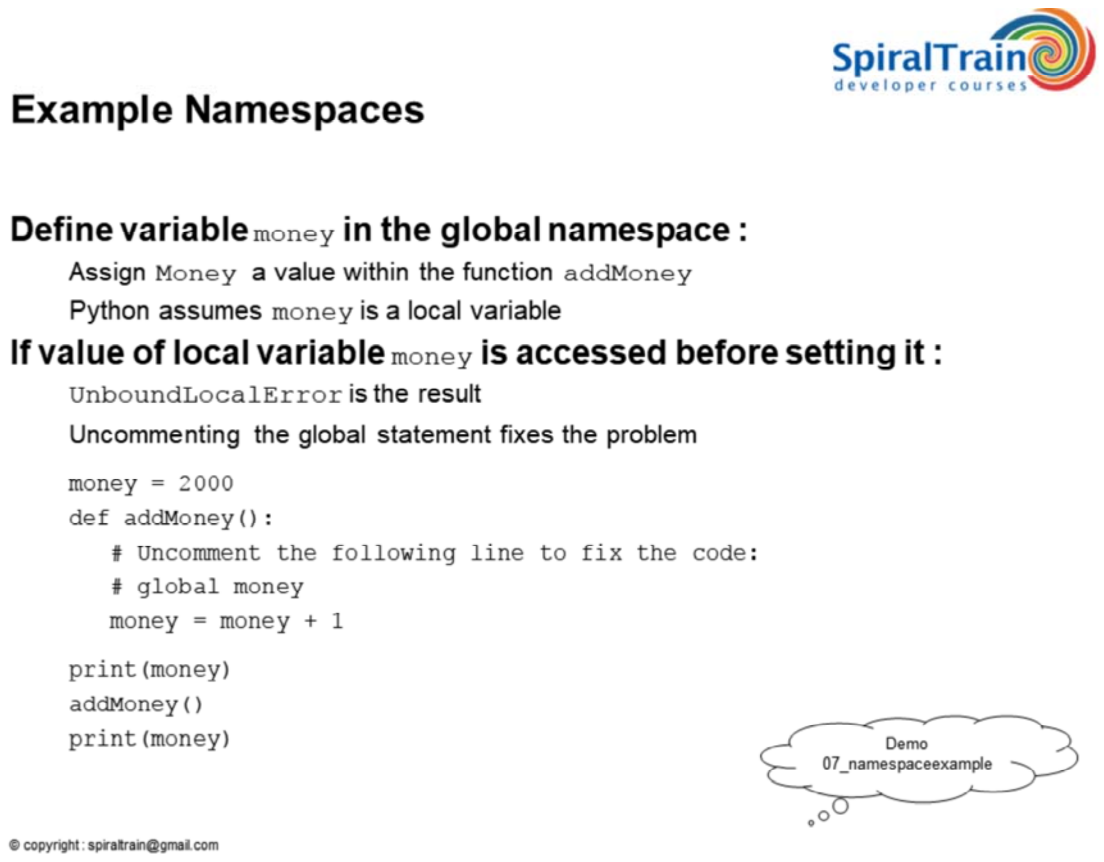
© copyright : spiraltrain@gmail.com

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that’s normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as abs(), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function maximize without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression z.real, real is an attribute of the object z. Strictly speaking, references to names in modules are attribute references: in the expression modname.funcname, modname is a module object and funcname is an attribute of it. In this case there happens to be a straightforward mapping between the module’s attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write modname.the_answer = 42. Writable attributes may also be deleted with the delstatement. For example, del modname.the_answer will remove the attribute the_answer from the object named by modname.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called __main__, so they have their own global namespace. (The built-in names actually also live in a module; this is called builtins)



Example Namespaces

Define variable `money` in the global namespace :

- Assign `Money` a value within the function `addMoney`
- Python assumes `money` is a local variable

If value of local variable `money` is accessed before setting it :

- `UnboundLocalError` is the result
- Uncommenting the global statement fixes the problem

```
money = 2000
def addMoney():
   # Uncomment the following line to fix the code:
   # global money
   money = money + 1

print(money)
addMoney()
print(money)
```

Demo
07_namespaceexample

© copyright : spiraltrain@gmail.com

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:

the innermost scope, which is searched first, contains the local names

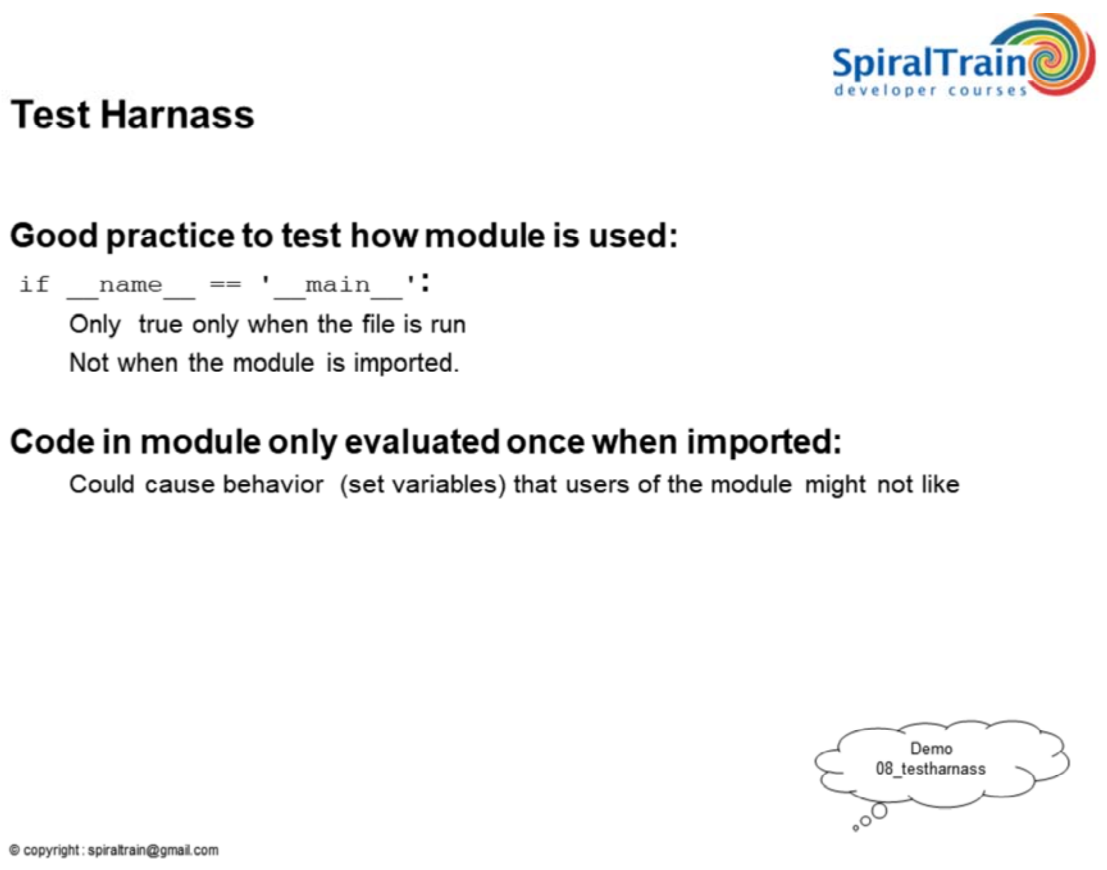
the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names

the next-to-last scope contains the current module’s global names

the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the middle scope containing the module’s global names. To rebind variables found outside of the innermost scope, the nonlocal statement can be used; if not declared nonlocal, those variable are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module’s namespace. Class definitions place yet another namespace in the local scope.



Test Harnass

Good practice to test how module is used:

```
if __name__ == '__main__':
```

Only true only when the file is run

Not when the module is imported.

Code in module only evaluated once when imported:

Could cause behavior (set variables) that users of the module might not like

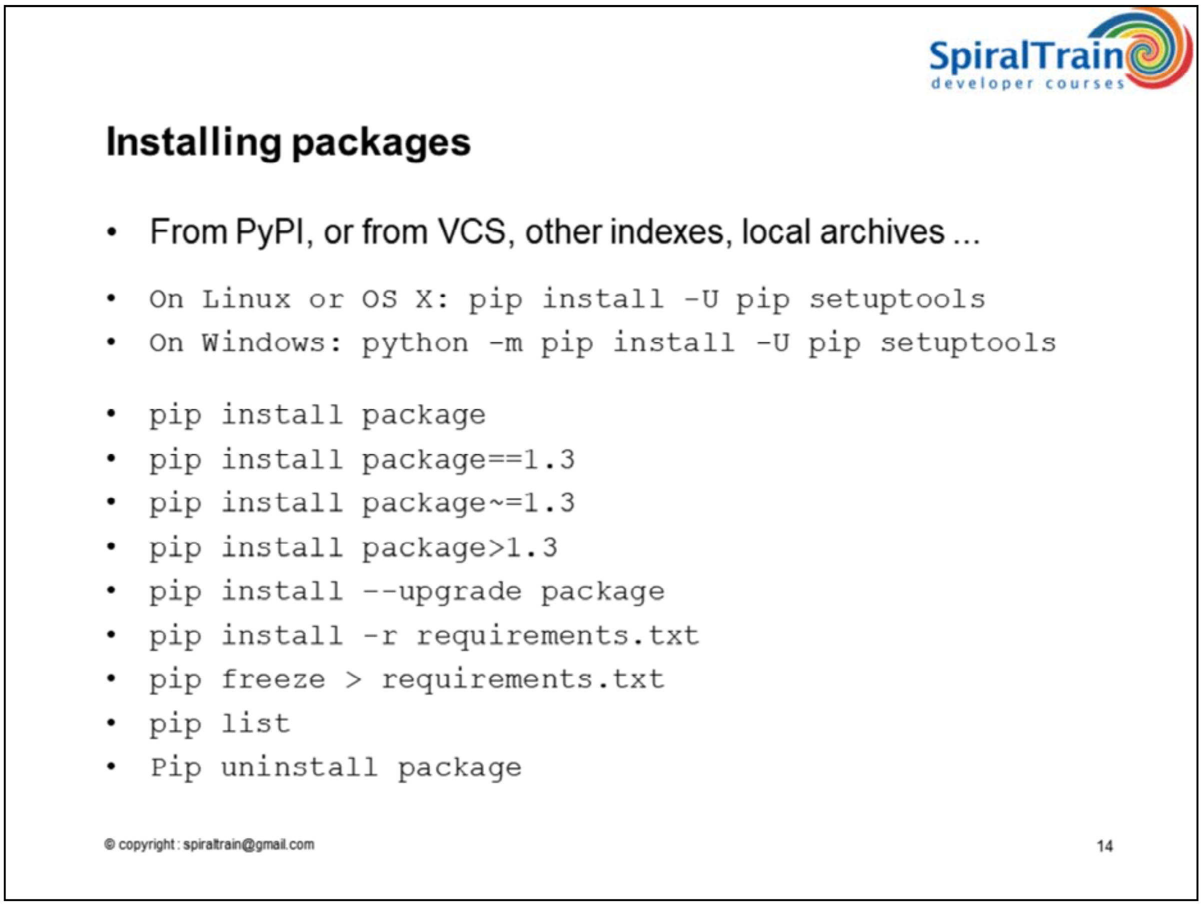
Demo
08_testharnass

© copyright : spiraltrain@gmail.com

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at "compile" time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no global statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement del x removes the binding of x from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, import statements and function definitions bind the module or function name in the local scope.

The global statement can be used to indicate that particular variables live in the global scope and should be rebound there; the nonlocal statement indicates that particular variables live in an enclosing scope and should be rebound there.



Installing packages

- From PyPI, or from VCS, other indexes, local archives ...

```
• On Linux or OS X: pip install -U pip setuptools
• On Windows: python -m pip install -U pip setuptools

• pip install package
• pip install package==1.3
• pip install package~=1.3
• pip install package>1.3
• pip install --upgrade package
• pip install -r requirements.txt
• pip freeze > requirements.txt
• pip list
• Pip uninstall package
```

© copyright : spiraltrain@gmail.com

14

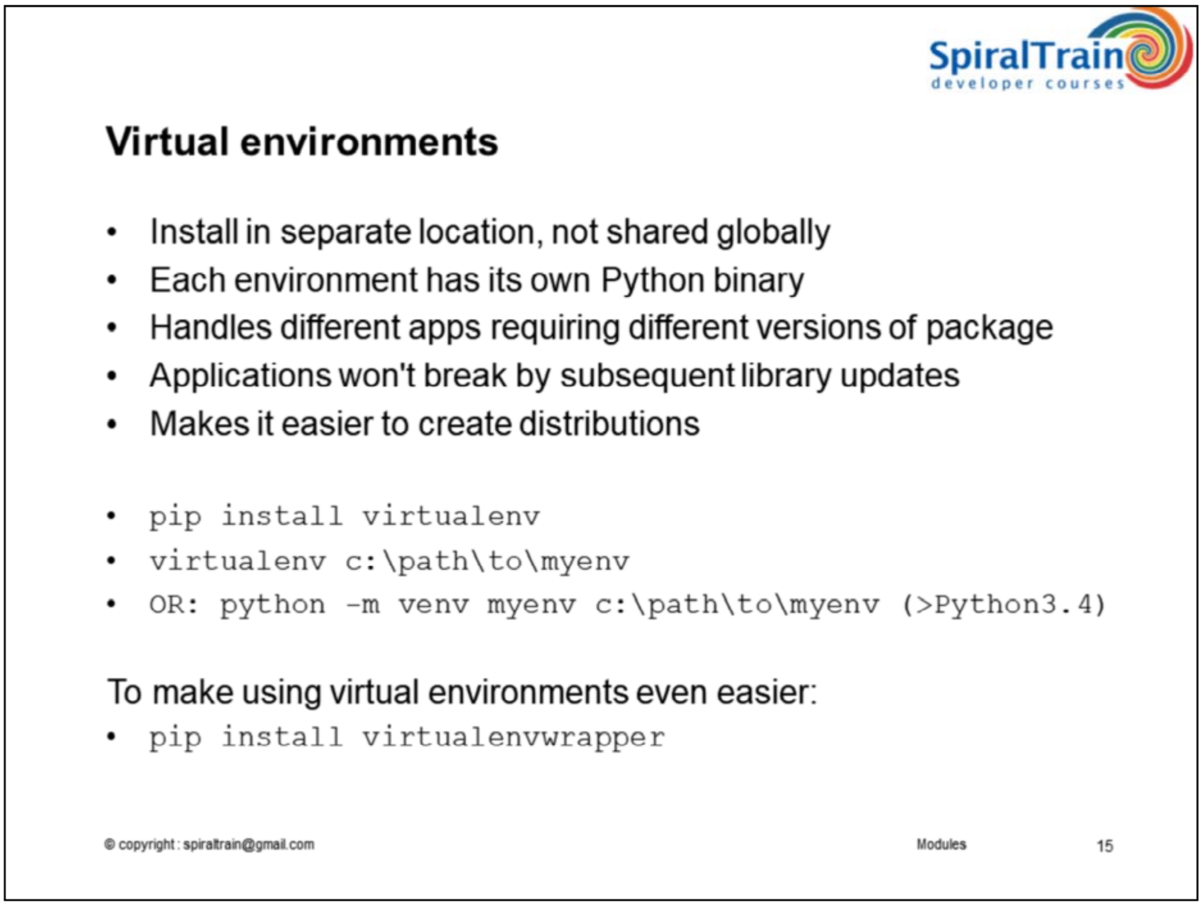
It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at "compile" time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no global statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement del x removes the binding of x from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, import statements and function definitions bind the module or function name in the local scope.

The global statement can be used to indicate that particular variables live in the global scope and should be rebound there; the nonlocal statement indicates that particular variables live in an enclosing scope and should be rebound there.

Slide 14

J1 site.getsitepackages()
Jos , 6/20/2017



Virtual environments

- Install in separate location, not shared globally
- Each environment has its own Python binary
- Handles different apps requiring different versions of package
- Applications won't break by subsequent library updates
- Makes it easier to create distributions

```
• pip install virtualenv
• virtualenv c:\path\to\myenv
• OR: python -m venv myenv c:\path\to\myenv (>Python3.4)
```

To make using virtual environments even easier:

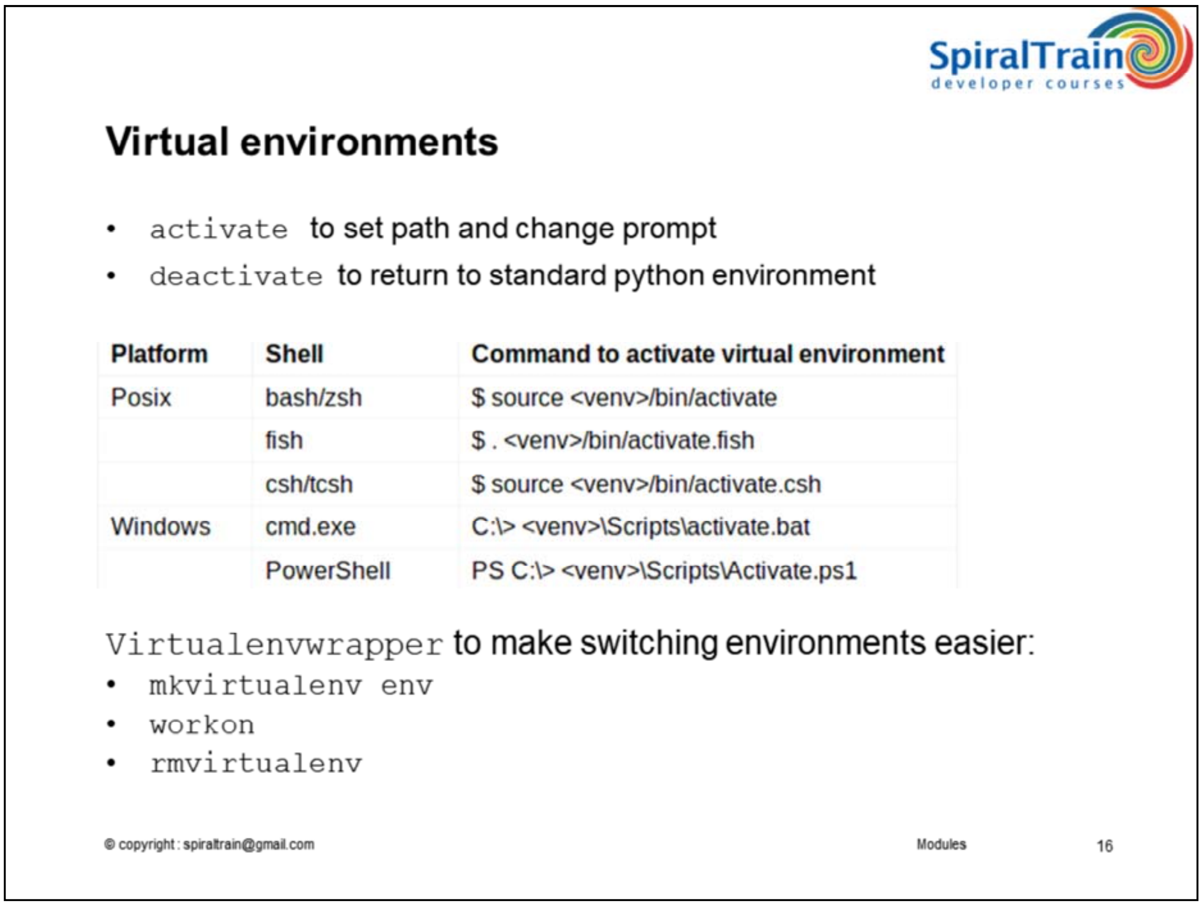
```
• pip install virtualenvwrapper
```

© copyright : spiraltrain@gmail.com

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module’s namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don’t rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no global statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement del x removes the binding of x from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, import statements and function definitions bind the module or function name in the local scope.

The global statement can be used to indicate that particular variables live in the global scope and should be rebound there; the nonlocal statement indicates that particular variables live in an enclosing scope and should be rebound there.



Virtual environments

- `activate` to set path and change prompt
- `deactivate` to return to standard python environment

Platform	Shell	Command to activate virtual environment
Posix	bash/zsh	$ source <venv>/bin/activate
	fish	$. <venv>/bin/activate.fish
	csh/tcsh	$ source <venv>/bin/activate.csh
Windows	cmd.exe	C:\> <venv>\Scripts\activate.bat
	PowerShell	PS C:\> <venv>\Scripts\Activate.ps1

`Virtualenvwrapper` to make switching environments easier:

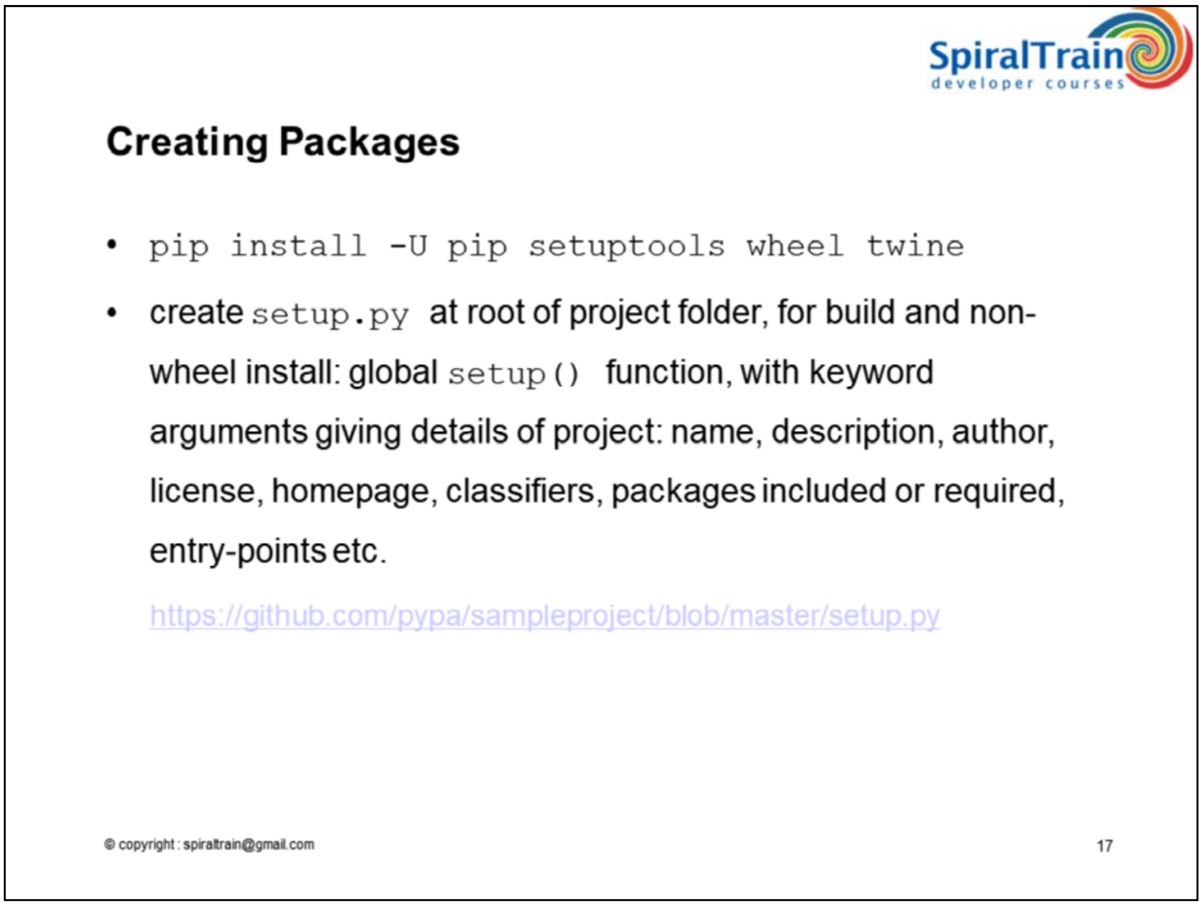
- `mkvirtualenv env`
- `workon`
- `rmvirtualenv`

© copyright : spiraltrain@gmail.com Modules 16

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at "compile" time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no global statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement del x removes the binding of x from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, import statements and function definitions bind the module or function name in the local scope.

The global statement can be used to indicate that particular variables live in the global scope and should be rebound there; the nonlocal statement indicates that particular variables live in an enclosing scope and should be rebound there.



Creating Packages

- `pip install -U pip setuptools wheel twine`
- create `setup.py` at root of project folder, for build and non-wheel install: global `setup()` function, with keyword arguments giving details of project: name, description, author, license, homepage, classifiers, packages included or required, entry-points etc.

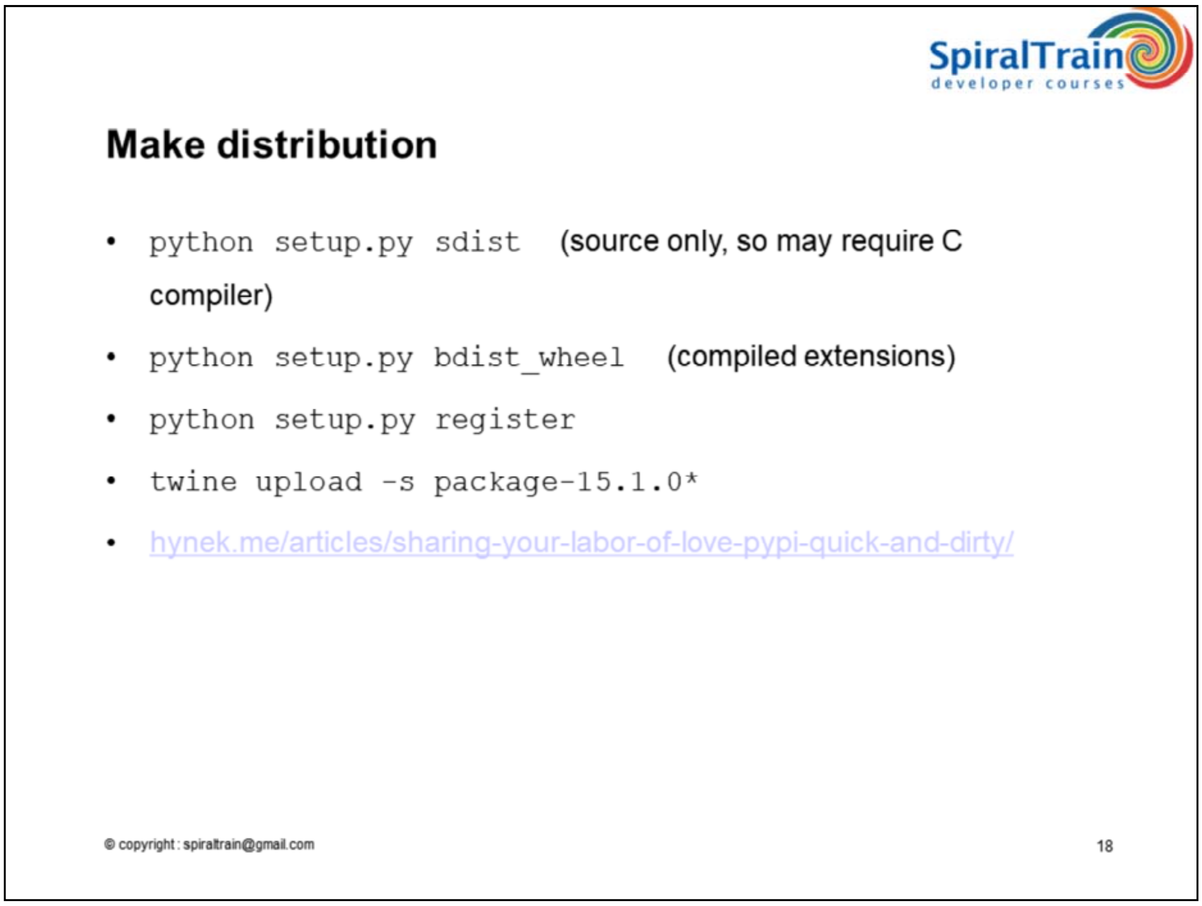
  https://github.com/pypa/sampleproject/blob/master/setup.py

© copyright : spiraltrain@gmail.com

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module’s namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don’t rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no global statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement del x removes the binding of x from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, import statements and function definitions bind the module or function name in the local scope.

The global statement can be used to indicate that particular variables live in the global scope and should be rebound there; the nonlocal statement indicates that particular variables live in an enclosing scope and should be rebound there.



Make distribution

- `python setup.py sdist` (source only, so may require C compiler)
- `python setup.py bdist_wheel` (compiled extensions)
- `python setup.py register`
- `twine upload -s package-15.1.0*`
- hynek.me/articles/sharing-your-labor-of-love-pypi-quick-and-dirty/

© copyright : spiraltrain@gmail.com

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at "compile" time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no global statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement del x removes the binding of x from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, import statements and function definitions bind the module or function name in the local scope.

The global statement can be used to indicate that particular variables live in the global scope and should be rebound there; the nonlocal statement indicates that particular variables live in an enclosing scope and should be rebound there.