PES UNIVERSITY

Department of Computer Science and Engineering (UG Studies)

PES University, Bangalore, India

Introduction to Computing using Python (UE19CS101)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

The import Statement

Module contents are made available to the caller with the import statement.

The import statement takes many different forms, shown below.

- `import <module_name>`

The simplest form is the one already shown above:

```
import   <module_name>
```

Note that this *does not* make the module contents *directly* accessible to the caller. Each module has its own **private symbol table**, which serves as the global symbol table for all objects defined *in the module*. Thus, a module creates a separate **namespace**, as already noted.
The statement **`import <module_name>`** only places **`<module_name>` in the caller's symbol table.** The *objects* that are defined in the module *remain in the module's private symbol table*.

From the caller, objects in the module are only accessible when prefixed with `<module_name>` via dot notation.

- `from <module_name> import <name(s)>`

An alternate form of the import statement allows individual objects from the module to be imported *directly into the caller's symbol table*:

```
from <module_name> import <name(s)>
```

Following execution of the above statement, <name(s)> can be referenced in the caller's environment without the <module_name> prefix.

Because this form of import places the object names directly into the caller's symbol table, any objects that already exist with the same name will be *overwritten*.

It is even possible to indiscriminately import everything from a module at one fell swoop:

- **`from <module_name> import *`**

This will place the names of *all* objects from <module_name> into the local symbol table, with the exception of any that begin with the underscore (_) character.
This isn't necessarily recommended in large-scale production code. It's a bit dangerous because you are entering names into the local symbol table *en masse*. Unless you know them all well and can be confident there won't be a conflict, you have a decent chance of overwriting an existing name inadvertently. However, this syntax is quite handy when you are just mucking around with the interactive interpreter, for testing or discovery purposes, because it quickly gives you access to everything a module has to offer without a lot of typing.

- **`from <module_name> import <name> as <alt_name>`**

It is also possible to import individual objects but enter them into the local symbol table with alternate names:

```
from <module_name> import <name> as <alt_name>[, <name> as <alt_name> …]
```

This makes it possible to place names directly into the local symbol table but avoid conflicts with previously existing names.

- **`import <module_name> as <alt_name>`**

You can also import an entire module under an alternate name:

```
import <module_name> as <alt_name>
```

Module contents can be imported from within a function definition. In that case, the import does not occur until the function is *called*:

```
# mod.py
def foo(arg):
      print('arg = ',arg)

# moddemo.py
def bar():
        from mod import foo
        foo('corge')
bar()
```

Output:

```
arg = corge
```

However, **Python 3** does not allow the indiscriminate import * syntax from within a function.

The dir() Function

The built-in function dir() returns a list of defined names in a namespace. Without arguments, it produces an alphabetically sorted list of names in the current local symbol table:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

>>> qux = [1, 2, 3, 4, 5]
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'qux']

>>> class Bar():
...     pass
...
>>> x = Bar()
>>> dir()
['Bar', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'qux', 'x']
```

Note how the first call to dir() above lists several names that are automatically defined and already in the namespace when the interpreter starts. As new names are defined (qux, Bar, x), they appear on subsequent invocations of dir().

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError',
'BaseException','BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError',
'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError',
'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError',
'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError',
'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit',
'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '_',
'__build_class__', '__debug__', '__doc__', '__import__', '__name__', '__package__',
'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
```

```
 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict',
'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format',
'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map',
'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print',
'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```