Department of Computer Science and Engineering
PES University, Bangalore, India

Python For Computational Problem Solving (UE22CS151A)

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

Modules in Python

Modular programming

Modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or modules. Individual modules can then be cobbled together like building blocks to create a larger application.

There are several advantages to modularizing code in a large application:

1. Simplicity:
   - Rather than focusing on the entire problem at hand, a module typically focuses on one relatively small portion of the problem. If you're working on a single module, you'll have a smaller problem domain to wrap your head around. This makes development easier and less error-prone.

1. Maintainability:
   - Modules are typically designed so that they enforce logical boundaries between different problem domains. If modules are written in a way that minimizes interdependency, there is decreased likelihood that modifications to a single module will have an impact on other parts of the program. (You may even be able to make changes to a module without having any knowledge of the application outside that module.) This makes it more viable for a team of many programmers to work collaboratively on a large application.

1. Reusability:
   - Functionality defined in a single module can be easily reused (through an appropriately defined interface) by other parts of the application. This eliminates the need to duplicate code.

1. Scoping:
   - Modules typically define a separate namespace, which helps avoid collisions between identifiers in different areas of a program. (One of the tenets in the Zen of Python is Namespaces are one honking great idea—let's do more of those!)

Functions, modules and packages are all constructs in Python that promote code modularization.

Python Modules

- Python has a way to put definitions in a file and use them in a script or in an application. Such a file is called a module;
- definitions from a module can be imported into other modules or into the main module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing python statements, definitions and classes. The file name is the module name with the suffix .py appended.

A module is a file containing a set of related code, aimed at solving specific tasks.

Within a module, the module's name (as a string) is available as the value of the built-in global variable

```
__name__
```

A module is yet another python program file. The only requirement is that the filename without the extension should be an identifier – should start with a letter of English and should be followed by letters of English, digits or underscores.

Then why do we require a module?

1. Share as a library:
   - If we write all our code in a single file, it cannot be used by others. So create the functions, classes in a separate file.
2. Develop libraries in a domain:
   - This helps modular development. We develop all routines related to a particular domain. We have already used such modules like math, os and so on.
   - Well known python Libraries: TensorFlow, Scikit-Learn, Numpy, Keras, PyTorch, LightGBM, Eli5, SciPy, pandas, matplotlib, nltk ....
3. Avoid name clashes:
   - The names used in a module do not normally clash with the names in another module.

Example:

In [ ]:
```
# filename: module1.py
print("hello from module1")  #all these statements are executed on import
print("How are you?")
```

Let us consider the file module1.py. As you can make out, it is just a regular python program.

In [1]:
```
# filename: ex1.py
# module:
#       1. physical unit of reuse
#       2. refers to a file
#       3. name should be an identifier
#       4. include using import statement
#       5. executed at the point during the program execution
#       6. not similar to #include of 'C' - executed before compilation
#       7. can be used to avoid clash of global names

print("one")
import module1  # Import statement causes execution of the module
import math     #   dynamically when that point is reached.
print("two")
print(module1)
print(math)
```

```
print(type(module1))
print("three")
```

```
one
hello from module1
How are you?
two
<module 'module1' from 'C:\\Users\\DELL\\Python Pgms - PCPS\\module1.py'>
<module 'math' (built-in)>
<class 'module'>
three
```

We can execute the file module1.py in two ways.

1. directly as python module1.py
2. Indirectly using import module1 in some other python program.

- Import statement causes execution of the module dynamically when that point is reached during the execution of the ex1.py file.
- Import is not like #include of some other language, which literally copies the file.
- Import does a few more things. We will explore all these as we go along.

Observe there is no change in the directory structure when we execute a python file directly.

On import, the python file is compiled to an intermediate form and is stored in a directory called **pycache** as a .pyc file. Next time we import the same file in the same or any other program, the compiled code is directly used.

Observe the time of modification of .pyc file and the corresponding .py file. If the .py file is changed and is imported, a new .pyc file is created. So, .pyc file is recreated only if required. This concept is called a build concept.

What is a .pyc file?

- A .pyc file is created by the Python translator when a .py file is imported into any other python file.
- The .pyc file contains the "compiled bytecode" of the imported module/program so that the "translation" from source code to bytecode can be skipped on subsequent imports of the .py file.
- Once the .pyc file is generated, there is no need for the .py file, unless the .py file is modified.
- The compiled .pyc files are placed in a **pycache** subdirectory and are named differently depending on which Python interpreter created them.

In [ ]:
```
# filename: module2.py

import math

def area_rect(l, b):
    return l * b

def area_triangle(a, b, c):
    s = (a + b + c) / 2.0
    area = math.sqrt(s * (s - a) * (s - b) * (s - c))
    return area

PI = 3.14
def area_circle(r):
    return PI * r * r
```

In [3]:
```
# filename : ex2.py
```

```
# can import number of modules using comma separated list

import module2

# find area of rectangle
#print("area : ", area_rect(20, 10)) # error

# should use a fully qualified name
print("Area of Rectangle : ", module2.area_rect(20, 10))
print("Area of Triangle : ", module2.area_triangle(3, 4, 5))
print("Area of Circle : ", module2.area_circle(7))
print(module2.PI)
#print(dir())
```

```
Area of Rectangle :  200
Area of Triangle :  6.0
Area of Circle :  153.93791
3.14159
```

In [1]:
```
import  math
print(dir())
```

```
['In', 'Out', '_', '__', '___', '__builtin__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', '_dh', '_i', '_i1', '_ih', '_ii', '_iii', '_oh',
'exit', 'get_ipython', 'math', 'quit']
```

Let us now examine module2.py and ex2.py.

The program file module2.py contains three functions to find areas of rectangle, triangle and circle. It also has a hard coded value of PI.

The user of the program ex2.py imports the module module2.py. He cannot access either functions or the variables of the module directly. He can access them using the fully qualified name as

```
modulename.functionname(<arg>)
```

There is no clash of names. The client can also have a variable called PI or a function called area_rect.

The import Statement

Module contents are made available to the caller with the import statement. The import statement takes many different forms.

The different ways of importing modules.

1) import <module_name>

In [ ]:
```
#Example:
import module2

# This statement,
 # 1. does not make the module contents directly accessible to the caller
 # 2. places the module-name in the caller's symbol_table/namespace
# In this case, use fully qualified name to access any module member.

# The global namespace contains any names defined at the level of the main program.
```

In [2]:
```
print(dir()) # global namespace
```

```
['In', 'Out', '_', '__', '___', '__builtin__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', '_dh', '_i', '_i1', '_i2', '_ih', '_ii', '_iii',
'_oh', 'exit', 'get_ipython', 'module2', 'quit']
```

In [ ]:
```
print(dir(__builtin__)) # builtin namespace
```

In [ ]:
```
print(dir(math)) # math module namespace
```

the above import statement causes module2 to become an entity of type module. We can access any member in module2 using fully qualified name. There is no clash of names.

In [1]:
```
# 1) import <module_name>
import module2
#print(module2)
#print(type(module2))
print("PI =", module2.PI)
```

```
PI = 3.14159
```

```
Note that the statement

import <module_name>

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, as illustrated above.

After the below import statement, module2 is placed into the local symbol table. Thus, module2 has
meaning in the caller's local context:
```

In [2]:
```
import module2
print(dir())
```

```
['In', 'Out', '_', '__', '___', '__builtin__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', '_dh', '_i', '_i1', '_i2', '_ih', '_ii', '_iii',
'_oh', 'exit', 'get_ipython', 'math', 'module1', 'module2', 'quit']
```

To avoid using long module names, the user can alias them to a shorter name as in the following example.

2) import <module_name> as <short_name>

Example:

import module2 as m2

From this point, m2 is an alias for module2 and any of them can be used interchangeably.

In [6]:
```
# 2. the user can create shorter and/or meaningful name

import module2 as m2
print(m2, type(m2))
print("PI ", m2.PI)
print("Area of Rectangle : ", m2.area_rect(20, 10))
#print(dir())
```

```
<module 'module2' from 'C:\\Users\\DELL\\Python Pgms - PCPS\\module2.py'> <class 'modul
e'>
```

```
PI  3.14159
Area of Rectangle :  200
```

To avoid using qualified long names, we may want to import a list of names from the module.

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

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

Example:

from module2 import PI, area_rect

In this case, the module name is no more available as a module type – so fully qualified name will not work. This way of import may cause name clashes.

In [3]:
```
# 3. selective import of symbols

from module2 import PI, area_rect
# module2 is no more a Pythonic entity
#print(module2, type(module2))

# can access the members without qualifying
# cannot access those which are not imported
#print("area : ", module2.area_rect(20, 10)) # error

def area_rect(l, b):
    print('******')
    return l * b

print("area : ", area_rect(20, 10))
print("area : ", module2.area_rect(20, 10))

#print("area : ", module2.area_circle(7)) # error
#print("area : ", area_circle(7)) # error
print()
print(dir())
```

```
******
area :  200

['In', 'Out', 'PI', '_', '__', '___', '__builtin__', '__builtins__', '__doc__', '__loade
r__', '__name__', '__package__', '__spec__', '_dh', '_i', '_i1', '_i2', '_i3', '_ih', '_
ii', '_iii', '_oh', 'area_rect', 'exit', 'get_ipython', 'math', 'quit']
```

A few questions for you to think.

1. What if an imported function calls the other not imported function?
2. What if there is a function with the same name defined?
   - before import or
   - after import

To avoid using qualified long names, we may want to import all names from a module to the current python program.

4) from module2 import *

*stands for all the names.

In [8]:
```
# 4. import all from module
```

```
from module2 import *
#print(module2, type(module2)) # error
print("area : ", area_rect(20, 10))
```

```
area :  200
```

This is the least preferred way of importing.

In [1]:
```
# 5. module10.py

def inc1(x):
    return x+1
```

In [2]:
```
# 5. module20.py

def inc1(x):
    return x+2
```

In [10]:
```
#from module10 import inc1
from module20 import inc1
#print(dir())
print(inc1(5))
```

```
7
```

We may execute the module directly or indirectly through the import mechanism. We may to do some testing when we execute directly which are not required when executed through the import mechanism.

How do we programmatically distinguish direct execution and execution through the import mechanism?

```
A module has a builtin variable whose name is __name__.

This variable has the name of the module on import and has the value '__main__'
on direct execution.

In the module, we use an idiom
if __name__ == '__main__':
and put the code to be executed on direct execution as the suite of the above
selection.
```

Example

Module: module4.py

In [4]:
```
# filename: module4.py

def foo():
    print("foo called")

def bar():
    print("bar called")

foo()
bar()
```

```
foo called
bar called
```

Will these methods (foo() and bar()) be called when we

```
    * execute this file directly: $python module4.py
```

- import in another file: $python client1_module4.py

Answer: YES

In [6]:
```
# filename: module4.py

def foo():
    print("foo called")

def bar():
    print("bar called")

foo()
bar()
```

```
foo called
bar called
```

In [1]:
```
# client1_module4.py
import module4
```

```
foo called
bar called
```

How do we avoid executing this test code on import?

In [2]:
```
# client1_module4.py
from module4 import foo, bar
```

How to distinguish between direct execution and import?

In [ ]:
```
Check the variable:  __name__
    * on direct execution : '__main__'
    * on import : 'module4' # module name
```

In [5]:
```
# filename: module4.py

def foo():
    print("foo called")

def bar():
    print("bar called")

foo()
bar()
print("__name__: ", __name__)
```

```
foo called
bar called
__name__:  __main__
```

filename: module5.py

In [3]:
```
"""
Assuming this is file mymodule.py, then this string, being the
first statement in the file, will become the "mymodule" module's
docstring when the file is imported.
"""

def foo():
```

```
    """The foo function's docstring"""
    print("foo called")

def bar():
    """The bar function's docstring"""
    print("bar called")

print('__name__ = ',__name__)

if __name__ == '__main__':
    print('Called from module5.py: Direct execution')
    foo()
    bar()
```

```
__name__ =  __main__
Called from module5.py: Direct execution
foo called
bar called
```

In [3]:
```
# filename: module5_demo.py

import module5      # not getting output, restart the jupyter kernel.
                    # If module is modified, then to get the changes,
                    #   delete module5.cpython-38.pyc from __pycache__ folder
```

```
__name__ =  module5
```

The above import statement avoids the execution of methods (foo() and bar()) on import by putting the below statement in module5

In [ ]:
```
if __name__ == '__main__':
    foo()
    bar()
```

In [ ]:
```
# Below statement displays docstring of foo() function
```

In [9]:
```
? module5.foo
```

In [ ]:
```
# client1_module4.py
from module4 import foo, bar
```

Additional info:

If we create the modules in our directories, we cannot easily share these files. Can we import modules if the files are elsewhere? We can do that in two different ways.

In [ ]:
```
#filenme: MyLib/module5.py
#module5 in directory MyLib
def foo():
    print("hello from module5 of MyLib")


#filename: ex5_MyLib_module5.py
#import module5 # error
#module5.foo()

#import MyLib.module5 # error
#module5.foo()

import sys
print(sys.path)
sys.path.insert(0, '/home/module/MyLib/')
```

```
print(sys.path)
import module5
module5.foo()
```

- import checks for the filename in the current directory as well as all directories whose names are stored in a list variable sys.path.
- To import modules from some other directory, insert the path of the module in the variable sys.path.

Thats about modules.

What is a Namespace in Python?

A namespace is a collection of names. A namespace is a way of providing the unique name for each and every object in Python.

A namespace is a collection of currently defined symbolic names along with information about the object that each name references.

You can think of a namespace as a dictionary in which the keys are the object names and the values are the objects themselves. Each key-value pair maps a name to its corresponding object.

Different namespaces can co-exist at a given time but are completely isolated.

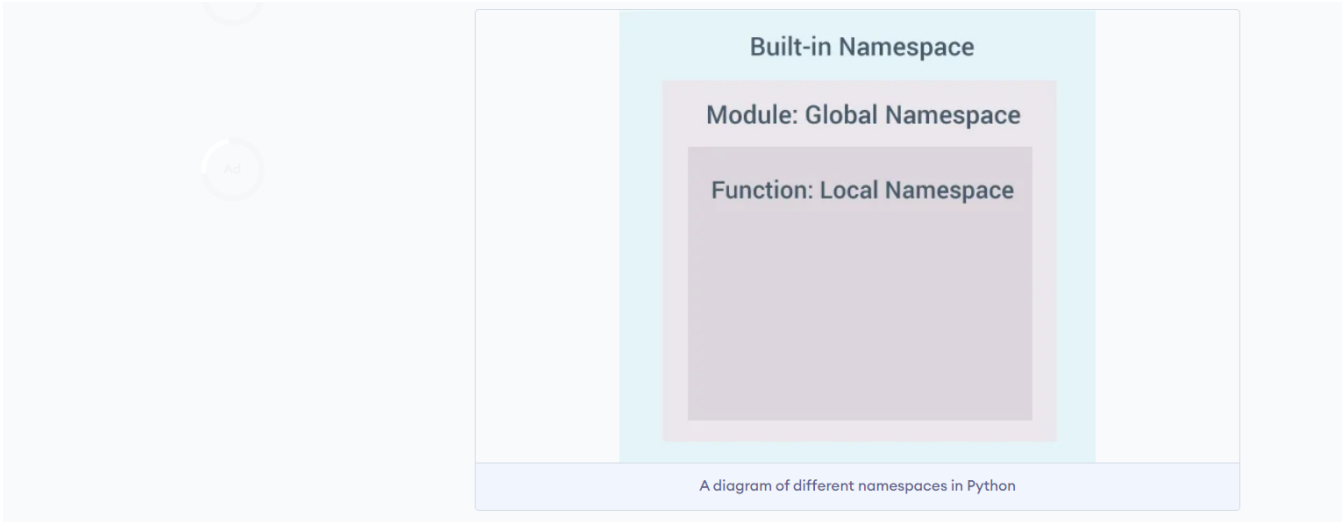
A namespace containing all the built-in names is created when we start the Python interpreter and exists as long as the interpreter runs.

This is the reason that built-in functions like id(), print() etc. are always available to us from any part of the program. Each module creates its own global namespace.

These different namespaces are isolated. Hence, the same name that may exist in different modules does not collide.

Modules can have various functions and classes. A local namespace is created when a function is called, which has all the names defined in it. Similar is the case with class. The following diagram may help to clarify this concept.

Nested Namespaces in Python Programming



In a Python program, there are four types of namespaces:

- Built-In
- Global
- Enclosing
- Local

These have differing lifetimes. As Python executes a program, it creates namespaces as necessary and deletes them when they're no longer needed. Typically, many namespaces will exist at any given time.

1) The Built-In Namespace

The built-in namespace contains the names of all of Python's built-in objects. These are available at all times when Python is running.

You can list the objects in the built-in namespace with the following command:

In [4]:
```
print(dir(__builtins__))
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOErro
r', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAb
ortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'Depre
cationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileE
xistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExi
t', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'Interr
uptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'Memor
yError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplement
ed', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'Pe
rmissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarni
ng', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxErr
or', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True',
'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeEr
ror', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
'WindowsError', 'ZeroDivisionError', '__IPYTHON__', '__build_class__', '__debug__', '__d
oc__', '__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'all',
'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'c
lassmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'di
splay', 'divmod', 'enumerate', 'eval', 'exec', 'filter', 'float', 'format', 'frozenset',
'get_ipython', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'i
nt', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'ma
x', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'proper
ty', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticm
ethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

You'll see some objects here that you may recognize — for example, the StopIteration exception, built-in functions like max() and len(), and object types like int and str.

The Python interpreter creates the built-in namespace when it starts up. This namespace remains in existence until the interpreter terminates.

2) The Global Namespace

- The global namespace contains any names defined at the level of the main program.
- Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates the python program.

Strictly speaking, this may not be the only global namespace that exists. The interpreter also creates a global namespace for any module that your program loads with the import statement.

In [1]:
```
import random
from math import sqrt,tan
```

```
name='Raj'
age=45
address='Bangalore'
print(dir())  #dir() without any argument shows what's in the global namespace.
```

```
['In', 'Out', '_', '__', '___', '__builtin__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', '_dh', '_i', '_i1', '_ih', '_ii', '_iii', '_oh',
'address', 'age', 'exit', 'get_ipython', 'name', 'quit', 'random', 'sqrt', 'tan']
```

To see the contents of any module namespace, you use dir(module_name).

In [3]:
```
import math
print(dir(math))
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asi
n', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degre
es', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isna
n', 'isqrt', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'perm',
'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'ta
u', 'trunc']
```

In [4]:
```
import math
print(math.pi)
```

```
3.141592653589793
```

Note that you write math.pi and not just simply pi. In addition to being a module, math acts as a namespace that keeps all the attributes of the module together. Namespaces are useful for keeping your code readable and organized.

3) The Local and Enclosing Namespaces

As you learned in the previous classes on functions, the interpreter creates a new namespace whenever a function executes. That namespace is local to the function and remains in existence until the function terminates.

Functions don't exist independently from one another only at the level of the main program. You can also define one function inside another

In [1]:
```
def fn_f(age,name):
    print('Start fn_f()')
    str1='Hello'

    def fn_g():
        age1=age
        name1=name
        print('Start fn_g()')
        print('Namespace of fn_g(): ', end='')
        print(dir())    # prints local namespace of fn_g()
        print('End fn_g()')
        return

    fn_g()
    print('Namespace of fn_f(): ', end='')
    print(dir())    # prints local namespace of fn_f()
    print('End fn_f()')
    return

fn_f(25,'Raj')
print()
print(dir())  # prints global namespace
```

```
Start fn_f()
Start fn_g()
Namespace of fn_g(): ['age', 'age1', 'name', 'name1']
End fn_g()
Namespace of fn_f(): ['age', 'fn_g', 'name', 'str1']
End fn_f()

['In', 'Out', '_', '__', '___', '__builtin__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', '_dh', '_i', '_i1', '_ih', '_ii', '_iii', '_oh',
'exit', 'fn_f', 'get_ipython', 'quit']
```

When the main program calls f(), Python creates a new namespace for f(). Similarly, when f() calls g(), g() gets its own separate namespace. The namespace created for g() is the local namespace, and the namespace created for f() is the enclosing namespace.

Each of these namespaces remains in existence until its respective function terminates. Python might not immediately reclaim the memory allocated for those namespaces when their functions terminate, but all references to the objects they contain cease to be valid.

Variable Scope

The existence of multiple, distinct namespaces means several different instances of a particular name can exist simultaneously while a Python program runs. As long as each instance is in a different namespace, they're all maintained separately and won't interfere with one another.

But that raises a question: Suppose you refer to the name x in your code, and x exists in several namespaces. How does Python know which one you mean?

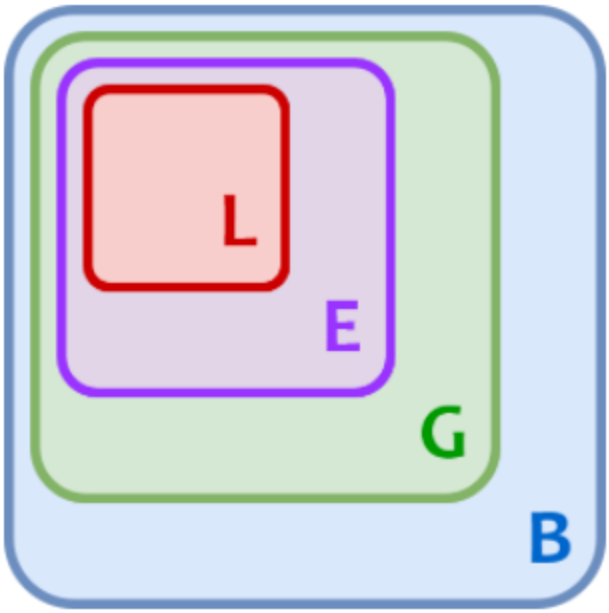
The answer lies in the concept of scope. The scope of a name is the region of a program in which that name has meaning. The interpreter determines this at runtime based on where the name definition occurs and where in the code the name is referenced.

To return to the above question, if your code refers to the name x, then Python searches for x in the following namespaces in the order shown:

1. Local:
    - If you refer to x inside a function, then the interpreter first searches for it in the innermost scope that's local to that function.
2. Enclosing:
    - If x isn't in the local scope but appears in a function that resides inside another function, then the interpreter searches in the enclosing function's scope.
3. Global:
    - If neither of the above searches is fruitful, then the interpreter looks in the global scope next.
4. Built-in:
    - If it can't find x anywhere else, then the interpreter tries the built-in scope.

This is the LEGB rule as it's commonly called in Python literature (although the term doesn't actually appear in the Python documentation).

The interpreter searches for a name from the inside out, looking in the local, enclosing, global, and finally the built-in scope:



If the interpreter doesn't find the name in any of these locations, then Python raises a NameError exception.

Python Packages

We usually organize our files in different folders and subfolders based on some criteria, so that they can be managed easily and efficiently.

For example, we keep all our games in a Games folder and we can even subcategorize according to the genre of the game or something like this. The same analogy is followed by the Python package.

- A Python module may contain several classes, functions, variables, etc.
- Whereas a Python package can contains several modules.
- In simpler terms a package is folder that contains various modules as files.

Creating Package

Let's create a package named mypckg that will contain two modules mod1 and mod2. To create this module follow the below steps:

In [ ]:
```
1. Create a folder named mypckg.
2. Inside this folder create an empty Python file i.e. __init__.py
3. Then create two modules mod1 and mod2 in this folder.
```

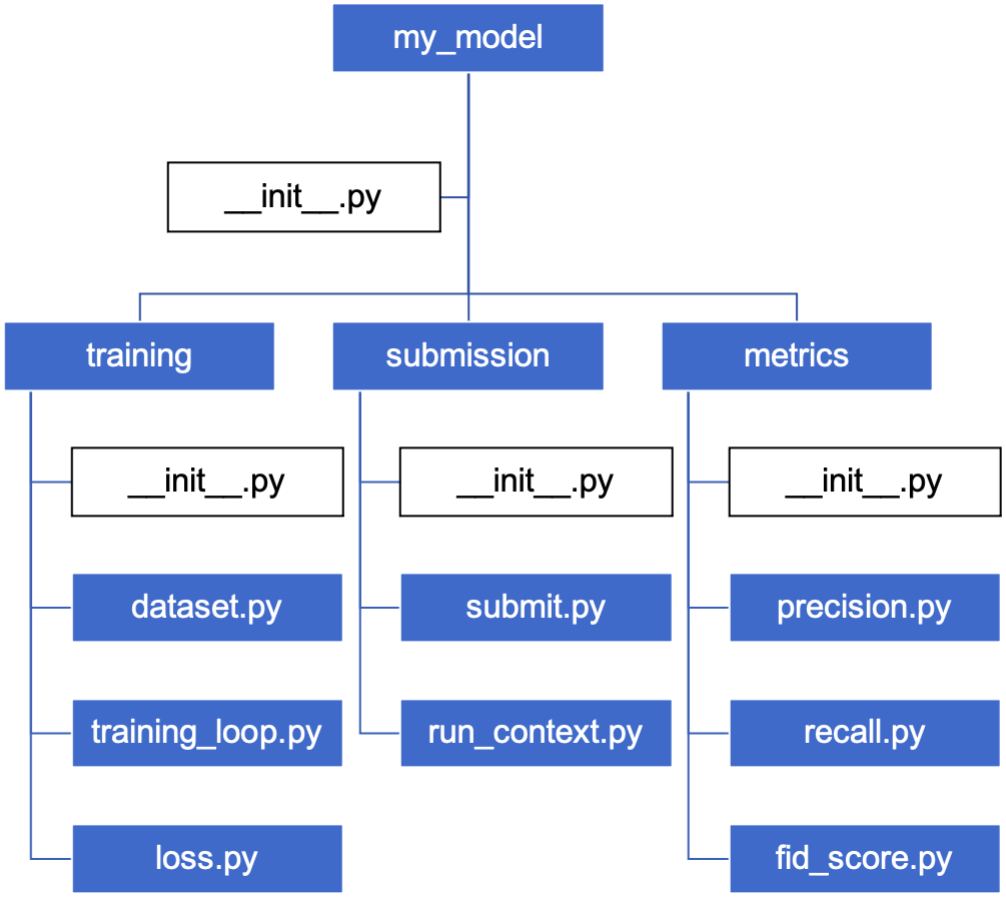
Each package should contain a file named **init**.py. This file usually includes the initialization code for the corresponding package.

Understanding **init**.py

- **init**.py helps the Python interpreter to recognise the folder as package.
- It also specifies the resources to be imported from the modules. If the **init**.py is empty this means that all the functions of the modules will be allowed to import.
- We can also specify the functions from each module to be made available.

Example:

Here's an example of the my_model package with three sub-packages: training, submission, and metrics.



To access code from a Python package, you can either import the entire package or its specific modules and sub-packages.

For example, to get access to the code defined in precision.py, you can:

- Import the whole package with import my_model;
- Import the metrics sub-package with import my_model.metrics;
- Import the precision.py module with either of these code snippets:

In [ ]:
```
import my_model.metrics.precision
# or
from my_model.metrics import precision
```

References:

1. 15_module.pdf – Prof. N S Kumar, Dept. of CSE, PES University.
2. https://www.w3schools.com/python
3. https://docs.python.org/3/tutorial/modules.html