Python advanced class

Module 1, The interpreter and runtime environment

- Command line parameters and environment variables
- Everything is an object
- Namespaces and scope
- The Garbage Collector

Command line parameters and environment variables

Running Python from command line

- Python runs as the interpreter from the commandline, even when runned from the editor like Visual Studio Code
- After the name of the script may come additional commandline parameters

```
shell> python myscript.py file1.txt file2.txt
```

- The execution depends on:
    - The location and version of Python
    - The current working directory
    - Environment variables

Important command line switches

- Python allows the use of more than 20 different commandline switches
- The most important ones are:
    - `-e` *commands* to run a set of commands from the commandline
    - `-m` *modulename* to execute a module in the module path
    - `-i` to continue into interactive mode after execution of script
    - `-V` prints the version of Python

parameters and sys.argv

- When additional parameters are passed on the commandline they are available in sys.argv

```
shell> python script.py file1.txt file2.txt
```

```
sys.argv == ['script.py', 'file1.txt', 'file2.txt']
```

Environment variables

- The `PATH` variable determines which Python enterpreter is used
- This also affects which `site-lib` directory is searched for third-party packages
- Additional modules and packages can be added through the `PYTHONPATH` variable
- From the Python script the environment variables can be accessed through `os.environ`

argsparse

- To help build good CLI tools `argparse` makes it easy to handle switches and parameters

```
import argparse
parser = argparse.ArgumentParser(description="Calculates the least 
common multiple of A and B")
parser.add_argument('--verbose', '-v', help='Verbose output', 
action="store_true")
parser.add_argument('A', help='First number', type=int)
parser.add_argument('B', help='Second number', type=int)
args = parser.parse_args()
```

Everything is an object

Mutable vs immutable objects

- Every data, function, class, mechanism, and operation is an object
- Some objects are immutable like `int`, `str`, `tuple`, and `bool`, which means that they can't be changed
- Some objects like `list`, `dict`, and `set` can be changed by adding, removing, or changing elements

In [1]:
```
L1 = [1, 2, 3]
S1 = "abc"
L1[0] = 11
print(L1)
try:
    S1[0] = "A"
except TypeError:
    print("Strings are immutable")
```

```
[11, 2, 3]
Strings are immutable
```

Alias vs copy

- Assignments is the operation of binding a variable (or attribute) to an object
- Objects are not dublicated by assigments
- Many classes implement a `.copy()` method to make a (shallow) copy
- The `copy` module provides a `deepcopy()` function to make an independent copy

In [2]:
```
L1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
L2 = L1           # Alias
L3 = L1.copy()    # Shallow copy
from copy import deepcopy
L4 = deepcopy(L1) # Deep copy
print(id(L1), id(L2), id(L3), id(L4))
print(id(L1[0]), id(L2[0]), id(L3[0]), id(L4[0]))
```

```
135428279396992 135428279396992 135428279397824 135428279396224
135428279397440 135428279397440 135428279397440 135428279400576
```

Attributes

- Attributes are "things" associated with an object written after a dot `.`
- For modules it is classes, functions, and variables in the module namespace
- For classes it is methods and variables of the class
- For objects it is class- or instance- methods or variables of the instance

In [3]:
```
import math
print(math.pi)
print(int.__itemsize__)
print([1, 2, 1].count(1))
```

```
3.141592653589793
4
2
```

Dunder variables and methods

- Functions and variables begining with two underscores and ending with two underscores are called "dunder" symbols
- "dunder" symbols have a pre-defined meaning given by the runtime environment
- It is not adviced to add new "dunder" symbols without good knowledge of the development of Python

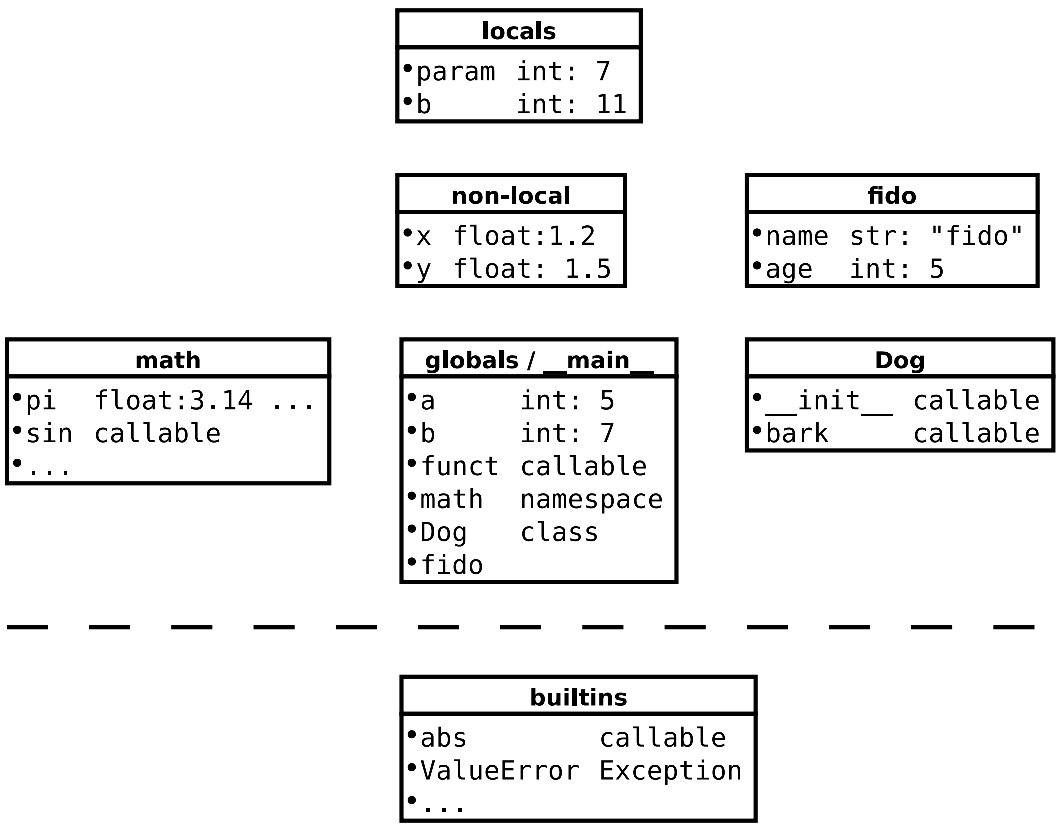
In [4]:
```
print(__name__)
print([1, 2, 3].__class__)
print("abcdefg".__len__())
```

```
__main__
<class 'list'>
7
```

Namespaces and scope

Namespaces and symboltables

- All symbols (variables, functions, classes, ...) in Python resides in a symboltable
- A symboltable is like a dictionary with variable name as the key and the object as a value
- A namespace is a naming context and owns the symboltable
- As we shall see, there are at least seven diffent types of namespaces



The global namespace

- All variables assigned outside of a function or a class belong to the global namespace
- The global namespace has a lifespan of the full execution time of the script
- The global namespace can be accessed as a `dict` by the function `globals()`

In [5]:
```
x = 5
g = globals()
print(g['x'])
```

```
5
```

Local namespaces

- A local namespace is created each time a function is called
- All parameters and variables assigned inside the function call belong to the local namespace
- If a function needs to assign a global variable, it must be declared global in the top of the function
- The local namespace is destroyed when the function returns or is abandonned from an exception

In [6]:
```
x = 5
def funct (param): # The param is a local variable
    global gl      # The variable gl is a global variable
    loc = x        # loc is a local variable, x is a global variable
    gl = x * param # The global variable gl is assigned
    return loc + gl
```

Module namespace

- When a module is imported by `import module` the symbols of the modules are placed in its own namespace
- This prevents a phenomena called *namespace pollution*
- The module namespace is also a global namespace and has a lifespan of the full execution time of the script

In [7]:
```
import math
print(math.pi)
```

```
3.141592653589793
```

Classes and objects namespaces

- Classes has their own namespace of methods and class attributes
- Each instance of a class also has its own namespace for instance variables
- When a symbol is accessed on an object, first the instance symboltable is searched, then the class

In [8]:
```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def bark(self):
        print(f"{self.name} says Wouf!")
fido = Dog("fido", 5)
fido.bark()
```

```
fido says Wouf!
```

Built-ins

- Every function, type, exception, and other symbol of the system has a namespace `__builtins__`
- If a built-in function is shadowed by a global symbol of the same name, then the function can still be accessed
- `dir(__builtins__)` will list all symbols of the built-in namespace

In [9]:

```
print(dir(__builtins__)[104:112])
abs = "Absolute"
print(__builtins__.abs(-14))
del abs
```

```
['display', 'divmod', 'enumerate', 'eval', 'exec', 'execfile',
'filter', 'float']
14
```

Eval and Exec functions

- The functions `eval` and `exec` can request to the interpreter to parse a text string and run it
- To be able to run code that potentially may be unsafe, they both use their own set of global and local namespaces

In [10]:

```
gl = {'a': 5, 'b': 7}
loc = {}
print(eval("a * b", gl))
codeblock = """
c = a * 3 + b * 7
a = -75
"""
exec(codeblock, gl, loc)
print(gl['a'], loc)
```

```
35
5 {'c': 64, 'a': -75}
```

The garbage collector

How the garbage collector works

- The garbage collector keeps track of memory use and re-claim dead objects.
- An object is considered dead if it is inaccessable directly og indirectly through active namespaces
- It applies two different schemes to detect inaccessable object:
    - **Reference Count** where each object is counted up when linked from a symboltable or becoming member of e.g. a `list`
    - **Mark and Sweep** where all names in symboltables are followed to object while all accessed objects are marked. Those not marked are the considered inaccessable.

The gc module

- It is possible to interact with the garbage collector through the `gc` module
- This could be desirable to postpone garbage collection to after a specific operation

In [11]:
```
import gc
gc.disable()    # No garbage collection is done from here
# Some operation
gc.collect()    # Run garbage collection now!
gc.enable()     # Enable automatic garbage collection again
```