301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

AP-26*: Functions, Decorators and OOP*

We have seen:

- Installing Python & main documentation
- Useful commands
- Modules: importing and executing
- Basics of the language
- Sequence datatypes
- Dictionaries
- Boolean expressions
- Control flow
- List Comprehension

Next topics

- Function definition
- Positional and keyword arguments of functions
- Functions as objects
- Higher-order functions
- Namespaces and Scopes
- Object Oriented programming in Python
- Inheritance
- Iterators and generators

Functions in Python - Essentials

- Functions are first-class objects
- All functions return some value (possibly **None**)
- Function call creates a new namespace
- Parameters are passed by object reference
- Functions can have optional keyword arguments
- Functions can take a variable number of args and kwargs
- Higher-order functions are supported

Function definition (1)

- Positional/keyword/default parameters

```
def sum(n,m):
    """ adds two values """
    return n+m

>>> sum(3,4)
7
>>> sum(m=5,n=3) # keyword parameters
8

#--------------------------------------

def sum(n,m=5): # default parameter
    """ adds two values, or increments by 5 """
    return n+m

>>> sum(3)
8
```

Function definition (2)

- Arbitrary number of parameters (varargs)

```
def print_args(*items): # arguments are put in a tuple
    print(type(items))
    return items


>>> print_args(1,"hello",4.5)
<class 'tuple'>
(1, 'hello', 4.5)

#--------------------------------------

def print_kwargs(**items): # args are put in a dict
    print(type(items))
    return items


>>> print_kwargs(a=2,b=3,c=3)
<class 'dict'>
{'a': 2, 'b': 3, 'c': 3}
```

Functions are objects

- As everything in Python, also functions are object, of class **function**

```
def echo(arg): return arg
type(echo)           # <class 'function'>
hex(id(echo))        # 0x1003c2bf8
print(echo)          # <function echo at 0x1003c2bf8>
foo = echo
hex(id(foo))         # '0x1003c2bf8'
print(foo)           # <function echo at 0x1003c2bf8>
isinstance(echo, object)       # => True
```

Function documentation

- The comment after the functions header is bound to the **__doc__** special attribute

```
def my_function():
    """Summary line: do nothing, but document it.
    Description: No, really, it doesn't do anything.
    """
    pass


print(my_function.__doc__)
# Summary line: Do nothing, but document it.
#
#     Description: No, really, it doesn't do anything.
```

Higher-order functions

- Functions can be passed as argument and returned as result
- Main combinators (**map**, **filter**) predefined: allow standard functional programming style in Python
- Heavy use of iterators, which support laziness
- Lambdas supported for use with combinators

```
lambda arguments: expression
```

  - The body can only be a single expression

Map

```
>>> print(map.__doc__)  % documentation
map(func, *iterables) --> map object
Make an iterator that computes the function using
arguments from each of the iterables.  Stops when the
shortest iterable is exhausted.
```

```
>>> map(lambda x:x+1, range(4))      % lazyness: returns
<map object at 0x10195b278>          % an iterator
>>> list(map(lambda x:x+1, range(4)))
[1, 2, 3, 4]
>>> list(map(lambda x, y : x+y, range(4), range(10)))
[0, 2, 4, 6]         % map of a binary function
>>> z = 5            % variable capture
>>> list(map(lambda x : x+z, range(4)))
[5, 6, 7, 8]
```

Map and List Comprehension

- **List comprehension** can replace uses of **map**

```
>>> list(map(lambda x:x+1, range(4)))
[1, 2, 3, 4]
>>> [x+1 for x in range(4)]
[1, 2, 3, 4]
>>> list(map(lambda x, y : x+y, range(4), range(10)))
[0, 2, 4, 6]    % map of a binary function
>>> [x+y for x in range(4) for y in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5,...  % NO!
>>> [x+y for (x,y) in zip(range(4),range(10))] % OK
[0, 2, 4, 6]
>>> print(zip.__doc__)
zip(iter1 [,iter2 [...]]) --> zip object
Return a zip object whose .__next__() method returns a tuple where
the i-th element comes from the i-th iterable argument. The
.__next__() method continues until the shortest iterable in the
argument sequence is exhausted and then it raises StopIteration.
```

Filter (and list comprehension)

```
>>> print(filter.__doc__)  % documentation
filter(function or None, iterable) --> filter object
Return an iterator yielding those items of iterable for
which function(item) is true. If function is None,
return the items that are true.
```

```
>>> filter(lambda x : x % 2 == 0,[1,2,3,4,5,6])
<filter object at 0x102288a58>  % lazyness
>>> list(_)                      % '_' is the last value
[2, 4, 6]
>>> [x for x in [1,2,3,4,5,6] if x % 2 == 0]
[2, 4, 6] % same using list comprehension
% How to say "false" in Python
>>> list(filter(None,
       [1,0,-1,"","Hello",None,[],[1],(),True,False]))
[1, -1, 'Hello', [1], True]
```

More modules for functional programming in Python

- **functools**: Higher-order functions and operations on callable objects, including:
  - reduce(*function*, *iterable*[, *initializer*])
- **itertools**: Functions creating ***iterators*** for efficient looping. Inspired by constructs from APL, Haskell, and SML.
  - count(10) --> 10 11 12 13 14 ...
  - cycle('ABCD') --> A B C D A B C D ...
  - repeat(10, 3) --> 10 10 10
  - takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
  - accumulate([1,2,3,4,5]) --> 1 3 6 10 15

Decorators

- A **decorator** is any callable Python object that is used to modify a **function**, **method** or **class definition**.
- A decorator is passed the original object being defined and returns a modified object, which is then bound to the name in the definition.
- (Function) Decorators exploit Python **higher-order features**:
  - Passing functions as argument
  - Nested definition of functions
  - Returning function
- Widely used in Python (system) programming
- Support several features of meta-programming

Basic idea: wrapping a function

```
def my_decorator(func):        # function as argument
    def wrapper(): # defines an inner function
        print("Something happens before the function.")
        func() # that calls the parameter
        print("Something happens after the function.")
    return wrapper # returns the inner function
```

```
def say_hello():   # a sample function
    print("Hello!")
# 'say_hello' is bound to the result of my_decorator
say_hello = my_decorator(say_hello) # function as arg
>>> say_hello()   # the wrapper is called
Something happens before the function.
Hello!
Something happens after the function.
```

Syntactic sugar: the "pie" syntax

```
def my_decorator(func):          # function as argument
    def wrapper(): # defines an inner function
        ... # as before
    return wrapper # returns the inner function
```

```
def say_hello():            ## HEAVY! 'say_hello' typed 3x
    print("Hello!")
say_hello = my_decorator(say_hello)
```

- Alternative, equivalent syntax

```
@my_decorator
def say_hello():
    print("Hello!")
```

Another decorator: do_twice

```
def do_twice(func):
    def wrapper_do_twice():
        func()      # the wrapper calls the
        func()      #      argument twice
    return wrapper_do_twice
```

```
@do_twice            # decorate the following
def say_hello():     # a sample function
    print("Hello!")
>>> say_hello()      # the wrapper is called
Hello!
Hello!
```

```
@do_twice           # does not work with parameters!!
def echo(str):      # a function with one parameter
    print(str)
>>> echo("Hi...")  # the wrapper is called
TypErr: wrapper_do_twice() takes 0 pos args but 1 was given
>>> echo()
TypErr: echo() missing 1 required positional argument: 'str'
```

do_twice for functions with parameters

- Decorators for functions with parameters can be defined exploiting ***args** and ****kwargs**

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_do_twice
```

```
@do_twice
def say_hello():
    print("Hello!")
>>> say_hello()
Hello!
Hello!
```

```
@do_twice
def echo(str):
    print(str)
>>> echo("Hi... ")
Hi...
Hi...
```

General structure of a decorator

- Besides passing arguments, the wrapper also forwards the **result** of the decorated function
- Supports introspection redefining **`__name__`** and **`__doc__`**

```
import functools
def decorator(func):
    @functools.wraps(func)     #supports introspection
    def wrapper_decorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_decorator
```

Example: Measuring running time

```
import functools
import time

def timer(func):
    """Print the runtime of the decorated function"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()
        value = func(*args, **kwargs)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value
    return wrapper_timer

@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])
```

Other uses of decorators

- Debugging: prints argument list and result of calls to decorated function
- Registering plugins: adds a reference to the decorated function, without changing it
- In a web application, can wrap some code to check that the user is logged in
- @staticmethod and @classmethod make a function invocable on the class name or on an object of the class
- More: decorators can be nested, can have arguments, can be defined as classes…

Example: Caching Return Values

```
import functools
from decorators import count_calls

def cache(func):
    """Keep a cache of previous function calls"""
    @functools.wraps(func)
    def wrapper_cache(*args, **kwargs):
        cache_key = args + tuple(kwargs.items())
        if cache_key not in wrapper_cache.cache:
            wrapper_cache.cache[cache_key] = func(*args, **kwargs)
        return wrapper_cache.cache[cache_key]
    wrapper_cache.cache = dict()
    return wrapper_cache

@cache
@count_calls   # decorator that counts the invocations
def fibonacci(num):
    if num < 2:
        return num
    return fibonacci(num - 1) + fibonacci(num - 2)
```

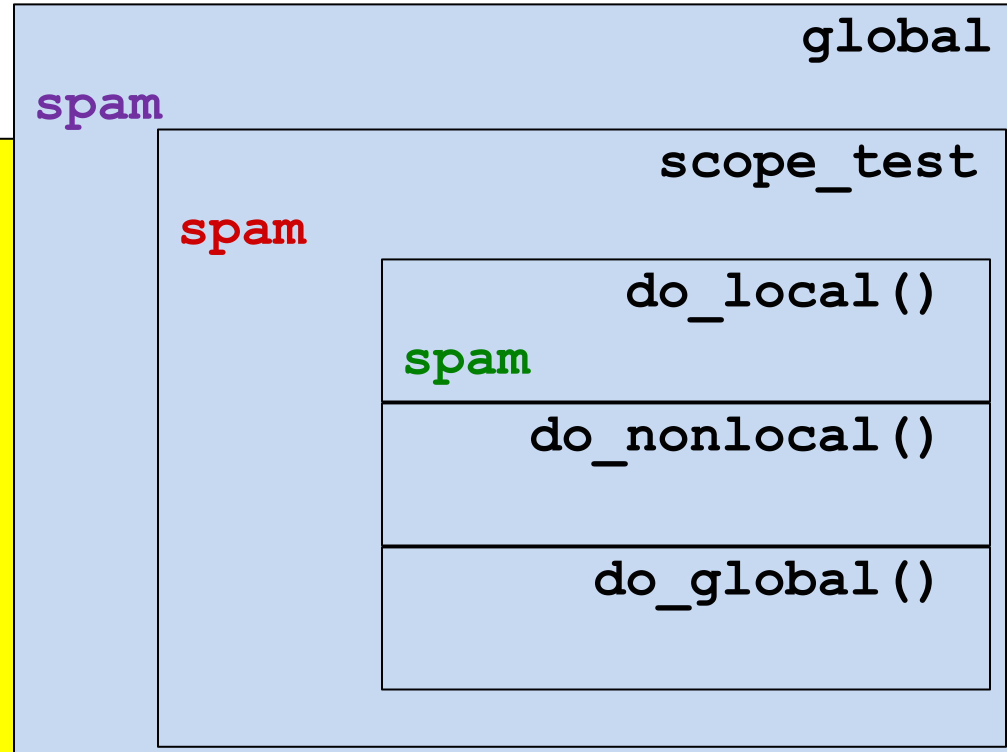
Namespaces and Scopes

- A ***namespace*** is a mapping from names to objects: typically implemented as a dictionary. Examples:
  - **builtins**: pre-defined functions, exception names,…
    - Created at intepreter's start-up
  - global names of a module
    - Created when the module definition is read
    - Note: names created in interpreter are in module __main__
  - local names of a function invocation
    - Created when function is called, deleted when it completes
  - and also names of a class, names of an object… see later
- Name ***x*** of a module ***m*** is an *attribute of* ***m***
  - accessible (read/write) with *"qualified name"* ***m.x***
  - if writable, it can be deleted with ***del***

Namespaces and Scopes (2)

- A **scope** is a textual region of a Python program where a namespace is **directly accessible**, i.e. reference to a name attempts to find the name in the namespace.
- Scopes are determined statically, but are used dynamically.
- During execution at least three namespaces are directly accessible, searched in the following order:
  - the scope containing the **local names**
  - the scopes of any enclosing functions, containing **non-local**, but also **non-global names**
  - the next-to-last scope containing the current module’s **global names**
  - the outermost scope is the namespace containing **built-in names**
- **Assignments to names go in the local scope**
- Non-local variables can be accessed using nonlocal or global

Scoping rules




```
def scope_test():

    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"

    do_local()
    print("After local assignment:", spam)      # not affected
    do_nonlocal()
    print("After nonlocal assignment:", spam)  # affected
    do_global()
    print("After global assignment:", spam)     # not affected

scope_test()
print("In global scope:", spam)
```

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Criticisms to Python: scopes

- Control structures don't introduce a new scope

```
def test():
  for a in range(5):
    b = a % 3
    print(b)
  print(b)

>>> test()
```

```
def test(x):
  print(x)
  for x in range(5):
    print(x)
  print(x)

>>> test("Hello!")
```

Closures in Python

- Python supports closures: Even if the scope of the outer function is reclaimed on return, the non-local variables referred to by the nested function are saved in its attribute `__closure__`

```
def counter_factory():
    counter = 0
    def counter_increaser():
        nonlocal counter
        counter = counter + 1
        return counter
    return counter_increaser

>>> f = counter_factory()
>>> f()
1
>>> f()
2
>>> f.__closure__
(<cell at 0x1033ace88: int object at 0x10096dce0>,)
```

OOP in Python

- Typical ingredients of the Object Oriented Paradigm:
  - **Encapsulation**: dividing the code into a public **interface**, and a private **implementation** of that interface;
  - **Inheritance**: the ability to create **subclasses** that contain specializations of their parent classes.
  - **Polymorphism**: The ability to **override** methods of a Class by extending it with a subclass (inheritance) with a more specific implementation (**inclusion polymorphism**)

From https://docs.python.org/3/tutorial/classes.html:

- *"Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation."*

Defining a class (object)

- A class is a blueprint for a new data type with specific internal ***attributes*** (like a struct in C) and internal functions (***methods***).
- To declare a class in Python the syntax is the following:

```
class className:
    <statement-1>
    …
    <statement-n>
```

- **`statements`** are assignments or function definitions
- A new namespace is created, where all names introduced in the statements will go.
- When the class definition is left, a ***class object*** is created, bound to **className**, on which two operations are defined: ***attribute reference*** and ***class instantiation***.
- *Attribute reference* allows to access the names in the namespace in the usual way

Example: Attribute reference on a class object

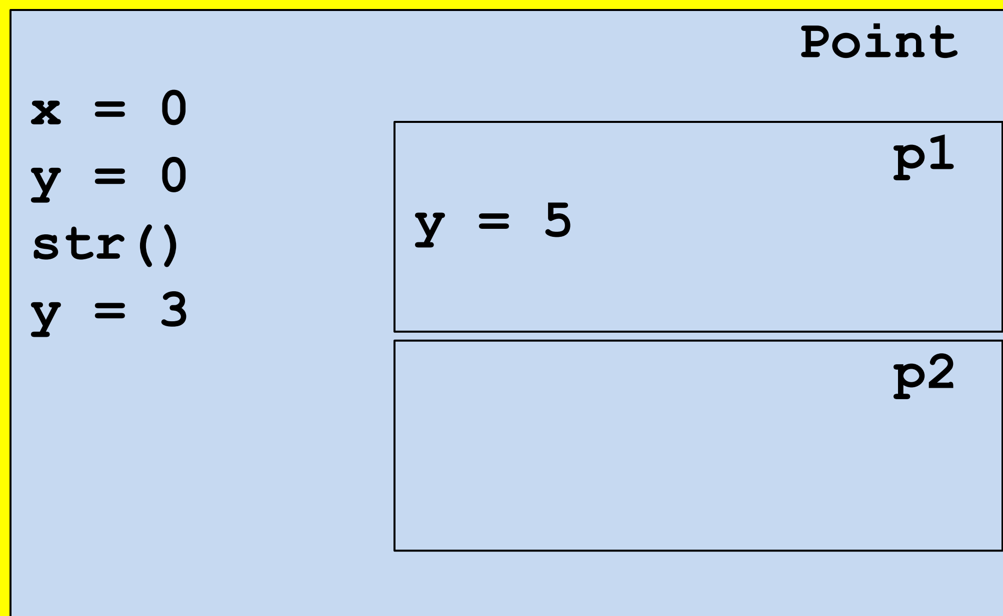
```
class Point:
  x = 0
  y = 0
  def str(): # no closure: needs qualified names to refer to x and y
      return  "x = " + (str) (Point.x) + ", y = " + (str) (Point.y)
#--------
import ...
>>> Point.x
0
>>> Point.y = 3
>>> Point.z = 5 # adding new name
>>> Point.z
5
>>> def add(m,n):
      return m+n
>>> Point.sum = add # adding new function
>>> Point.sum(3,4)
7
```

```
                Point
x = 0
y = 0
str()
y = 3
z = 5
sum = add(m,n)
```

Creating a class instance

- A **class instance** introduces a **new namespace *nested in the class namespace***: by visibility rules all names of the class are visible
- If no *constructor* is present, the syntax of class instantiation is **className():** the new namespace is empty

```
class Point:
  x = 0
  y = 0
  def str():
      return  "x = " + str(Point.x) + ", y = " + str(Point.y)
#--------
>>> p1 = Point()
>>> p2 = Point()
>>> p1.x
0
>>> Point.y = 3
>>> p2.y
3
>>> p1.y = 5
>>> p2.y
3
```



Instance methods

- A class can define a set of ***instance methods***, which are just functions:

```
def methodname(self, parameter1, ..., parametern):
        statements
```

- The first argument, usually called self, represents the ***implicit parameter*** (**`this`** in Java)
- A method ***must*** access the object's attributes through the **`self`** reference (eg. **self.x**) and the class attributes using **className.<attrName>** (or **self.__class__.<attrName>**)
- The first parameter must not be passed when the method is called. It is bound to the target object. Syntax:

```
obj.methodname(arg1, ..., argn):
```

- But it can be passed explicitly. Alternative syntax:

```
className.methodname(obj, arg1, ..., argn):
```

"Instance methods"

- Any function ***with at least one parameter*** defined in a class can be invoked on an instance of the class with the dot notation.

```
class Foo
    def fun(par-0, par-1, ..., par-n):
          statements
#----
>>>obj = Foo()
>>>obj.fun(arg-1,...,arg-n)
# is syntactic sugar for
>>>obj.__class__.fun(obj,arg-1,...,arg-n)
```

- Since the instance **`obj`** is bound to the first parameter, **`par-0`** is usually called **`self`**.
- A name **`x`** defined in the (namespace of the) instance is accessed as **`par-0.x`** (i.e., usually **`self.x`**)
- A name **`x`** defined in the class is accessed as **`className.x`** (or **`self.__class__.x`**)

Constructors

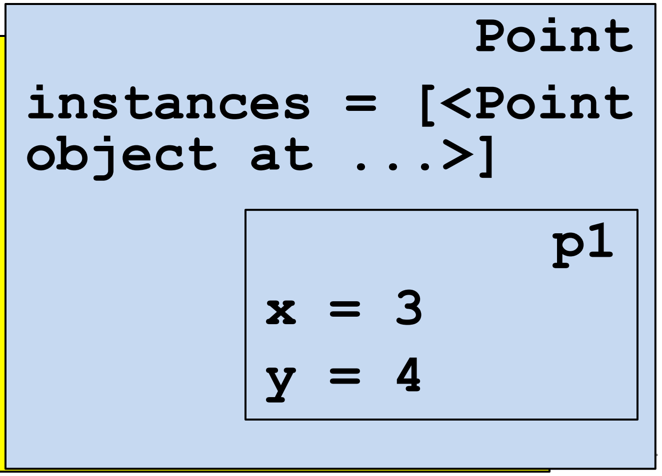
- A constructor is a **special instance method** with name `__init__`.

Syntax:

```
def __init__(self, parameter_1, ..., parameter_n):
        statements
```

- Invocation: `obj = className(arg_1, …, arg_n)`
- The first parameter `self` is bound to the new object.
- `statements` typically initialize (thus create) "instance variables", i.e. names in the new object namespace.
- Note: at most ONE constructor (**no overloading in Python**!)

```
class Point:
   instances = []
   def __init__(self, x, y):
      self.x = x
      self.y = y
      Point.instances.append(self)
#--------
>>> p1 = Point(3,4)
```

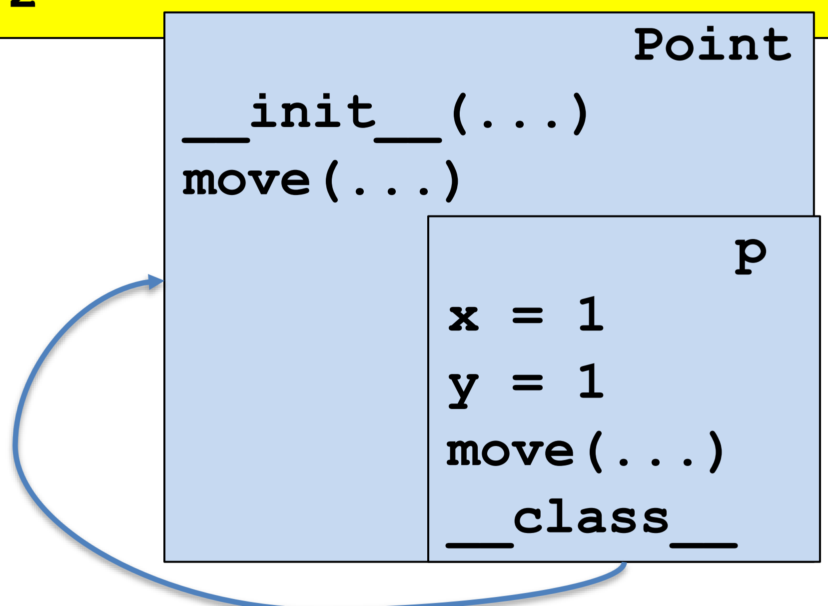


What about "methods in instances?"

- Instances are themselves namespaces: we can add functions to them.
- Applying the usual rules, they can hide "instance methods"

```
class Point:
   def __init__(self, x, y):
      self.x = x
      self.y = y
      def move(z,t):
         self.x -= z
         self.y -= t
      self.move = move
   def move(self,dx,dy):
      self.x += dx
      self.y += dy
```

```
>>> p = Point(1,1)
>>> p.x
1
>>> p.move(1,1)
>>> p.x
0
>>> p.__class__.move(p,2,2)
>>> p.x
2
```



String representation

- It is often useful to have a textual representation of an object with the values of its attributes. This is possible with the following instance method:

```
 def __str__(self) :
         return <string>
```

- This is equivalent to Java's **toString** (converts object to a string) and it is invoked automatically when **str** or **print** is called.

Special methods

- **Method overloading**: you can define special instance methods so that Python's built-in operators can be used with your class.

Binary Operators

Operator	Class Method
-	`__sub__(self, other)`
+	`__add__(self, other)`
*	`__mul__(self, other)`
/	`__truediv__(self, other)`

Operator	Class Method
==	`__eq__(self, other)`
!=	`__ne__(self, other)`
<	`__lt__(self, other)`
>	`__gt__(self, other)`
<=	`__le__(self, other)`
>=	`__ge__(self, other)`

Unary Operators

-	`__neg__(self)`
+	`__pos__(self)`

- Analogous to C++ overloading mechanism:
  - Pros: very compact syntax
  - Cons: may be more difficult to read if not used with care

(Multiple) Inheritance, in one slide

- A class can be defined as a *derived class*

```
class derived(baseClass):
    statements
    statements
```

- No need of additional mechanisms: the namespace of **derived** is nested in the namespace of **baseClass**, and uses it as the next non-local scope to resolve names
- All instance methods are automatically virtual: lookup starts from the instance (namespace) where they are invoked
- Python supports **multiple inheritance**

```
class derived(base1,..., basen):
    statements
    statements
```

- ***Diamond problem*** solved by an algorithm that linearizes the set of all (directly or indirectly) inherited classes: the **Method resolution order** (**MRO**) ➔ `ClassName.mro()`
- https://www.python.org/download/releases/2.3/mro/

Encapsulation (and "name mangling")

- **Private** instance variables (not accessible except from inside an object) **don't exist in Python.**
- **Convention:** a **name prefixed with underscore** (e.g. **`_spam`**) is treated as ***non-public part of the API*** (function, method or data member). It should be considered an implementation detail and subject to change without notice.

Name mangling ("storpiatura")

- Sometimes class-private members are needed to avoid clashes with names defined by subclasses. Limited support for such a mechanism, called *name mangling.*
- Any **name with at least two leading underscores and at most one trailing underscore** like e.g. **`__spam`** is textually replaced with **`_class__spam`**, where **`class`** is the current class name.

Example for name mangling

- Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls.

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update # private copy of update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Static methods and class methods

- **Static methods** are simple functions defined in a class with no **`self`** argument, preceded by the **@staticmethod** decorator
- They are defined inside a class but they cannot access instance attributes and methods
- They can be called through both the class and any instance of that class!
- **Benefits of static methods**: they allow subclasses to customize the static methods with inheritance. Classes can inherit static methods without redefining them.
- Class methods are similar to static methods but they have a first parameter which is the class name.
- Definition must be preceded by the **@classmethod** decorator
- Can be invoked on the class or on an instance.

Iterators

- An **iterator** is an object which allows a programmer to traverse through all the elements of a collection (**iterable** object), regardless of its specific implementation. In Python they are used implicitly by the **FOR** loop construct.

- Python iterator objects required to support two methods:
  - `__iter__` returns the iterator object itself. This is used in **FOR** and **IN** statements.
  - The `next` method returns the next value from the iterator. If there is no more items to return then it should raise a `StopIteration` exception.

- Remember that an iterator object can be used only once. It means after it raises `StopIteration` once, it will keep raising the same exception.

- Example:

```
for element in [1, 2, 3]:
      print(element)
```

```
>>> list = [1,2,3]
>>> it = iter(list )
>>> it
<listiterator object at 0x00A1DB50>
>>> it.next()
1
>>> it.next()
2
>>> it.next()
3
>>> it.next() -> raises StopIteration
```

Generators and coroutines

- **Generators** are a simple and powerful tool for creating iterators.
- They are written like **regular functions** but use the `yield` statement whenever they want to return data.
- Each time the `next()` is called, the generator resumes where it left-off (it remembers all the data values and which statement was last executed).
- ***Anything that can be done with generators can also be done with class based iterators (not vice-versa).***
- What makes generators so compact is that the `__iter__()` and `next()` methods are created automatically.
- Another key feature is that the local variables and execution state are **automatically saved** between calls.

Generators (2)

- In addition to automatic method creation and saving program state, when generators terminate, they automatically raise **`StopIteration`**.
- In combination, these features make it easy to create iterators with no more effort than writing a regular function.

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

#----------------

>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Typing in Python

- Dynamic, strong duck typing
- Code can be annotated with types

```
def greetings(name: str) -> str:
    return 'Hello ' + name.
```

- Module **typing** provides runtime support for type hints
- Type hints can be checked statically by external tools, like **mypy**
- They are ignored by CPython

Miscellaneous

- Overloading: forbidden, but not necessary
- Overriding: ok, thanks to namespaces
- Generics: type hints support generics

Garbage collection in Python

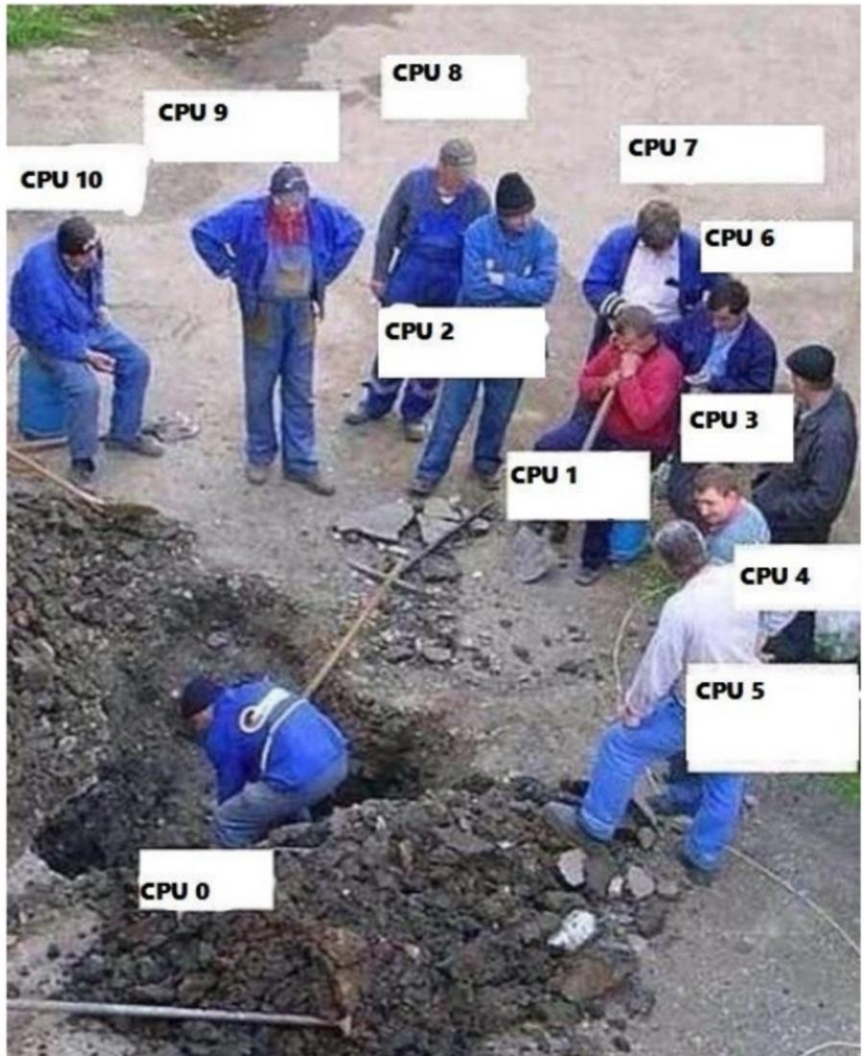
CPython manages memory with a **reference counting** + a **mark&sweep** cycle collector scheme

- Reference counting: each object has a counter storing the number of references to it. When it becomes 0, memory can be reclaimed.
- Pros: simple implementation, memory is reclaimed as soon as possible, no need to freeze execution passing control to a garbage collector
- Cons: additional memory needed for each object; cyclic structures in garbage cannot be identified (thus the need of mark&sweep)

Handling reference counters

- Updating the refcount of an object has to be done atomically
- In case of multi-threading you need to synchronize all the times you modify refcounts, or else you can have wrong values
- Synchronization primitives are quite expensive on contemporary hardware
- Since almost every operation in CPython can cause a refcount to change somewhere, handling refcounts with some kind of synchronization would cause *spending almost all the time on synchronization*
- As a consequence…

Concurrency in Python…



The Global Interpreter Lock (GIL)

- The CPython interpreter assures that only one thread executes Python bytecode at a time, thanks to the **Global Interpreter Lock**
- The current thread must hold the **GIL** before it can safely access Python objects
- This simplifies the CPython implementation by making the object model (including critical built-in types such as **dict**) implicitly safe against concurrent access
- Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, **at the expense of much of the parallelism afforded by multi-processor machines**.

More on the GIL

- However the GIL can degrade performance even when it is not a bottleneck. The system call overhead is significant, especially on multicore hardware.
- Two threads calling a function may take twice as much time as a single thread calling the function twice.
- The GIL can cause I/O-bound threads to be scheduled ahead of CPU-bound threads. And it prevents signals from being delivered.
- Some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing.
- Also, the GIL is always released when doing I/O.

Alternatives to the GIL?

- Past efforts to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case.
- It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.
- Guido van Rossum has said he will reject any proposal in this direction that slows down single-threaded programs.
- **Jython** (on JVM, -> 2017, Python 2.7) and **IronPython** (on .NET) have no GIL and can fully exploit multiprocessor systems
- **PyPy** (Python in Python, supporting JIT) currently has a GIL like CPython
- in **Cython** (compiled, for CPython extension modules) the GIL exists, but can be released temporarily using a "with" statement

Criticisms to Python: syntax of tuples

```
>>> type((1,2,3))
<class 'tuple'>
>>> type(())
<class 'tuple'>
>>> type((1))
<class 'int'>
>>> type((1,))
<class 'tuple'>
```

- Tuples are made by the commas, not by ( )
- With the exception of the empty tuple…

Criticisms to Python: indentation

- Lack of brackets makes the syntax "weaker" than in other languages: accidental changes of indentation may change the semantics, leaving the program syntactically correct.

```
def foo(x):
    if x == 0:
        bar()
        baz()
    else:
        qux(x)
        foo(x - 1)
```

```
def foo(x):
    if x == 0:
        bar()
        baz()
    else:
        qux(x)
    foo(x - 1)
```

- Mixed use of tabs and blanks may cause bugs almost impossible to detect

Criticisms to Python: indentation

- Lack of brackets makes it harder to refactor the code or insert new one
- "When I want to refactor a bulk of code in Python, I need to be very careful. Because if lost, I'm not sure what I'm editing belongs to which part of the code. Python depends on indentation, so if I have mistakenly removed some indentation, I totally have no idea whether the correct code should belong to that **if** clause or this **while** clause."
- Will Python change in the future?

```
>>> from __future__ import braces
  File "<stdin>", line 1
SyntaxError: not a chance
>>>
```

Builtins & Libraries

- The Python ecosystem is extremely rich and in fast evolution
- For available functions, classes and modules browse:
  - **Builtin Functions**
    - https://docs.python.org/3.8/library/functions.html
  - **Standard library**
    - https://docs.python.org/3.8/tutorial/stdlib.html
- There are dozens of other libraries, mainly for scientific computing, machine learning, computational biology, data manipulation and analysis, natural language processing, statistics, symbolic computation, etc.