PyLearn_PyTutorial_9_Classes

May 3, 2020

#

Learning Python

1 The Python Tutorial -> Classes

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

- Creating a new class creates a new type of object, allowing new instances of that type to be made.
- Each class instance can have **attributes** attached to it for maintaining its state. Class instances can also have **methods** (defined by its class) for modifying its state.
- It is a mixture of the class mechanisms found in C++ and Modula-3.
- 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.
- In C++ terminology, normally class members (including the data members) are public (except see below Private Variables), and **all member functions are virtual**.
- As in Modula-3, there are no shorthands for referencing the object's members from its methods:
  - the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call.
- As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming.
- Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user.
- Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

1.1 A Word About Names and Objects

- Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as **aliasing** in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types.
- This is usually used to the benefit of the program, since **aliases behave like pointers** in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and **if a function modifies an object passed as an argument, the caller will see the change** — this eliminates the need for two different argument passing mechanisms as in Pascal.

1.2 Python Scopes and Namespaces

- 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.
- 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!
- 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 `del` statement. 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`^^.)
- The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.
- 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 variables 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.
- 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`** or **`nonlocal`** 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.

1.2.1 Scopes and Namespaces Example

[2]:
```python
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)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

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
```

> Note : - the local assignment (which is default) didn't change scope_test's binding of spam,
> - the `nonlocal` assignment changed scope_test's binding of spam, and - the `global` assignment changed the module-level binding.
>
> You can also see that there was no previous binding for spam before the `global` assignment.

1.3 A First Look at Classes

- When a **class definition** is left normally (via the end), a **class object** is created.
- **Class objects** support two kinds of operations:
  - **attribute references**
  - **class instantiation**

Attribute References

[43]:
```
class MyClass:
    """A simple example class"""
    i = 12345

print(MyClass.__doc__)
print(MyClass.i)
```

```
A simple example class
12345
```

Class Instantiation

[46]:
```
class MyClass:
    """A simple example class"""
    i = 12345
```

```
m = MyClass()    # Class Instantiation
print(m.__doc__)
print(m.i)
```

```
A simple example class
12345
```

Examples

[51]:
```
class MyClass:
    """A simple example class"""   # docstring
    i = 12345

d = MyClass.__doc__  # docstring
j = MyClass.i
print(d)
print(j)

MyClass.__doc__ = "new docstring"
MyClass.i = 747
print(MyClass.__doc__)
print(MyClass.i)

m = MyClass()
print(m.__doc__)
print(m.i)
```

```
A simple example class
12345
new docstring
747
new docstring
747
```

[80]:
```
class MyClass1:
    def fun(self):      # needs to have self as first argument
        return 'hello MyClass1'

class MyClass2:
    def fun():
        return 'hello MyClass2'

m = MyClass1()
s = m.fun()
print(s)
```

```
m = MyClass1()
f = m.fun
s = f()      # ok
print(s)

m = MyClass2()
f = m.fun
s = f()       # ERROR!
print(s)
```

```
hello MyClass1
hello MyClass1


      ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-80-9aa35c691b6d> in <module>
        18 m = MyClass2()
        19 f = m.fun
   ---> 20 s = f()       # ERROR!
        21 print(s)


       TypeError: fun() takes 0 positional arguments but 1 was given
```

[50]:
```
class MyClass:
    def fun(self):
        return 'hello world'

m = MyClass()
s = m.fun()
print(s)

s = MyClass.fun(m)
print(s)
s = MyClass.fun(MyClass)
print(s)
g = MyClass.fun
print(g)
s = g(MyClass)
print(s)
```

```
def newfun(self):
    return "new function"

MyClass.fun = newfun

m = MyClass()
s = m.fun()
print(s)

s = MyClass.fun(m)
print(s)
s = MyClass.fun(MyClass)
print(s)
g = MyClass.fun
print(g)
s = g(MyClass)
print(s)
```

```
hello world
hello world
hello world
<function MyClass.fun at 0x7ff4f069eca0>
hello world
new function
new function
new function
<function newfun at 0x7ff4f06b99d0>
new function
```

[53]:
```
class MyList:
    """My own list class"""
    def __init__(self):
        self.data = []

    def add(self, item):
        self.data.append(item)

    def show(self):
        print(self.data)

mylist = MyList()
mylist.add(4)
mylist.add("rabbit")
mylist.add("apple")
mylist.show()
```

```
[4, 'rabbit', 'apple']
```

[60]:
```
class Complex:
    """My own complex number class"""
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
    def show(self):
        r = self.r
        i = self.i
        adder = " + "
        if i < 0:
            adder = " - "
            i = -i
        print("< " + str(r) + adder + str(i) + "i >")

c1 = Complex(3,4)
c1.show()
c2 = Complex(-8,2)
c2.show()
c3 = Complex(7,-1)
c3.show()
```

```
< 3 + 4i >
< -8 + 2i >
< 7 - 1i >
```

Instance Objects The only operations understood by **instance objects** are **attribute references**. There are two kinds of valid **attribute names**: - **data attributes** - correspond to “instance variables” in Smalltalk, and to “data members” in C++ - **methods** - function that “belongs to” an object

data attributes need not be declared; like local variables, they **spring into existence** when they are first assigned to.

[203]:
```
class MyClass:
    i = 321

m = MyClass()
print(m.i)

m.j = 564  # data attribute m.j spring into existence when it is first assigned␣
 ↪to
print(m.j)

del m.j    # can be deleted, reference to this after this would produce error

del m.i    # cannot be deleted
```

```
321
```

```
564
```

```
␣
↪---------------------------------------------------------------------------

    AttributeError                            Traceback (most recent call␣
↪last)

    <ipython-input-203-1c8047c279c5> in <module>
     10 del m.j    # can be deleted, reference to this after this would␣
↪produce error
     11
---> 12 del m.i    # cannot be deleted


    AttributeError: i
```

Method Objects Can be stored away and called at a later time.
Instance object is passed as the first argument of the function. #### IMPORTANT: The call `x.f()` is exactly equivalent to `MyClass.f(x)`.

[86]:
```python
class MyClass:
    def fun(self):      # needs to have self as first argument
        return 'hello MyClass'

m = MyClass()
s = m.fun()
print(s)

m = MyClass()
f = m.fun       # m.fun is a METHOD OBJECT !!
s = f()         # ok
print(s)

f = MyClass.fun # MyClass.fun is a FUNCTION OBJECT, but not METHOD OBJECT !!
s = f()         # ERROR!!
print(s)
```

```
hello MyClass
hello MyClass
```

```
␣
↪---------------------------------------------------------------------------
```

```
        TypeError                                 Traceback (most recent call␣
↪last)

        <ipython-input-86-5c9634146fc6> in <module>
         13
         14 f = MyClass.fun # MyClass.fun is a FUNCTION OBJECT, but not METHOD␣
↪OBJECT !!
    ---> 15 s = f()         # ERROR!!
         16 print(s)


        TypeError: fun() missing 1 required positional argument: 'self'
```

Class and Instance Variables **instance variables** are for data unique to each instance **class variables** are for attributes and methods shared by all instances of the class

[141]:
```python
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

    def f():
        kind = "mammal"

d = Dog('Fido')
e = Dog('Buddy')
print( d.kind )                 # shared by all dogs
print( e.kind )                 # shared by all dogs
print( d.name )                 # unique to d
print( e.name )                 # unique to e

d.kind = 'wolf'   # local version
print( d.kind )                 # returns 'wolf'
print( e.kind )                 # returns 'canine'
print( Dog.kind )               # returns 'canine'

del d.kind        # delete local version
print( d.kind )                 # returns 'canine'
print( e.kind )                 # returns 'canine'
print( Dog.kind )               # returns 'canine'
```

```
canine
canine
Fido
```

```
Buddy
wolf
canine
canine
canine
canine
canine
```

WRONG DESIGN!!

[95]:
```python
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
print( d.tricks )             # unexpectedly shared by all dogs
print( e.tricks )             # unexpectedly shared by all dogs
```

```
['roll over', 'play dead']
['roll over', 'play dead']
```

CORRECT DESIGN!!

[97]:
```python
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
print( d.tricks )
print( e.tricks )
```

```
['roll over']
```

```
['play dead']
```

1.4 Random Remarks

If the same attribute name occurs in both an instance and in a class, then **attribute lookup prioritizes the instance**

[98]:
```python
class Warehouse:
     purpose = 'storage'
     region = 'west'

w1 = Warehouse()
print(w1.purpose, w1.region)

w2 = Warehouse()
w2.region = 'east'
print(w2.purpose, w2.region)
```

```
storage west
storage east
```

- Data attributes may be referenced by methods as well as by ordinary users ("clients") of an object. In other words, **classes are not usable to implement pure abstract data types**. In fact, **nothing in Python makes it possible to enforce data hiding** — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)
- Clients should use data attributes with care — **clients may mess up invariants maintained by the methods** by stamping on their data attributes. Note that **clients may add data attributes of their own to an instance object without affecting the validity of the methods**, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.
- There is **no shorthand for referencing data attributes** (or other methods!) from within methods. I find that this actually increases the readability of methods: there is **no chance of confusing local variables and instance variables** when glancing through a method.
- Often, the first argument of a method is called **self**. This is **nothing more than a convention**: **the name self has absolutely no special meaning to Python**. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that **a class browser program might be written that relies upon such a convention**.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

[101]:
```python
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)
```

```
class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g

c = C()
print( c.f(5,-6) )
print( c.g() )
print( c.h() )
```

```
-1
hello world
hello world
```

> Now f, g and h are all **attributes** of class C that **refer to function objects**, and consequently they are all **methods of instances** of C

Methods may call other methods by using method attributes of the **self** argument:

[109]:
```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)

b = Bag()
b.add(4)
b.addtwice(6)
print(b.data)

print(Bag.__class__)
print(b.__class__)
```

```
[4, 6, 6]
<class 'type'>
<class '__main__.Bag'>
```

1.5 Inheritance

Given:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

- Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.
- There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.
- Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (**For C++ programmers: all methods in Python are effectively `virtual`.**)
- An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as `BaseClassName` in the global scope.)
- Python has two built-in functions that work with inheritance:
  - Use `isinstance()` to check an instance's type: `isinstance(obj, int)` will be `True` only if `obj.__class__` is `int` or some class derived from `int`.
  - Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(float, int)` is `False` since `float` is not a subclass of `int`.

1.5.1 Multiple Inheritance

Given:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

- For most purposes, in the simplest cases, you can think of the search for attributes inherited

from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in `DerivedClassName`, it is searched for in `Base1`, then (recursively) in the base classes of `Base1`, and if it was not found there, it was searched for in `Base2`, and so on.
- In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.
- Dynamic ordering is necessary because **all cases of multiple inheritance exhibit one or more diamond relationships** (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see https://www.python.org/download/releases/2.3/mro/.

1.6 Private Variables

- **"Private"** instance variables that cannot be accessed except from inside an object **don't exist in Python**. However, there is a **convention** that is followed by most Python code: a **name prefixed with an underscore** (e.g. _spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.
- Since there is a **valid use-case for class-private members** (namely to avoid name **clashes of names with names defined by subclasses**), there is limited support for such a mechanism, called **name mangling**. Any identifier of the form **`__spam`** (**at least two leading underscores**, **at most one trailing underscore**) is **textually replaced with `_classname__spam`**, where classname is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.
- **Name mangling** is helpful for letting subclasses override methods without breaking intra-class method calls. For example:

[147]:
```
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 original update() method
```

```python
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)

mylist1 = ["apple", "orange", "cucumber"]
map = MappingSubclass(mylist)
print(map.items_list)

mylist2 = ["apple", "orange", "cucumber"]
myquantity = [3, 5, 6]
map.update(mylist2, myquantity)
print(map.items_list)
```

```
['apple', 'orange', 'cucumber']
['apple', 'orange', 'cucumber', ('apple', 3), ('orange', 5), ('cucumber', 6)]
```

- The above example would work even if `MappingSubclass` were to introduce a `__update` identifier since it is replaced with `_Mapping__update` in the `Mapping` class and `_MappingSubclass__update` in the `MappingSubclass` class respectively.
- Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

[127]:
```python
class MyClass:
   i = 3
   _i = 4
   __i = 5

print(dir(MyClass))

print(MyClass.i)            # ok
print(MyClass._i)           # ok
print(MyClass._MyClass__i)  # name mingled

print(MyClass.__i)          # Will raise AttributeError
```

```
['_MyClass__i', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_i', 'i']
3
4
5
```

```
      ␣
↪---------------------------------------------------------------------------

       AttributeError                            Traceback (most recent call␣
↪last)

       <ipython-input-127-6e0d87e7d5f9> in <module>
         9 print(MyClass._i)           # ok
        10 print(MyClass._MyClass__i)  # name mingled
   ---> 11 print(MyClass.__i)  # name mingled


       AttributeError: type object 'MyClass' has no attribute '__i'
```

[148]:
```python
# Source: https://www.geeksforgeeks.org/name-mangling-in-python/
# Python code to illustrate how mangling works
# With method overriding

class Map:
    def __init__(self):
        self.__geek()

    def geek(self):
        print("In parent class")

    # private copy of original geek() method
    __geek = geek

class MapSubclass(Map):

    # provides new signature for geek() but
    # does not break __init__()
    def geek(self):
        print("In Child class")

# Driver's code
obj = MapSubclass()
obj.geek()
```

```
In parent class
In Child class
```

[156]:
```python
class A:
    i = 4
    _i = 40
    __i = 400

class B(A):
    i = 5
    _i = 40
    __i = 400

a = A()
b = B()

print(dir(A), "\n")
print(dir(a), "\n")

print(dir(B), "\n")
print(dir(b), "\n")
```

```
['_A__i', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_i', 'i']

['_A__i', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_i', 'i']

['_A__i', '_B__i', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_i', 'i']

['_A__i', '_B__i', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_i', 'i']
```

Take note the output on the mingled names: - `_A__i` for class `A` - `_A__i` and `_B__i` for class `B`

1.7 Odds and Ends

Sometimes it is useful to have a data type similar to the Pascal "record" or C "struct", bundling together a few named data items. An empty class definition will do nicely:

[110]:
```
class Employee:
    pass

john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

- A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.
- Instance method objects have attributes, too: **`m.__self__`** is the instance object with the method **`m()`**, and **`m.__func__`** is the function object corresponding to the method

[165]:
```
class A:
    def fun(self):
        print("hello")

    def funny(self):
        print("funny world")

a = A()
funobj = a.fun

print(funobj.__self__)
print(funobj.__func__)
s = funobj.__self__
f = funobj.__func__

f(a)
s.funny()
```

```
<__main__.A object at 0x7ff4f0625310>
<function A.fun at 0x7ff4f06825e0>
hello
funny world
```

1.8 Iterators

most container objects can be looped over using a `for` statement:

[167]:
```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("test_file.txt"):
    print(line, end='')
```

```
1
2
3
1
2
3
one
two
1
2
3
one
two
three
```

Content of test_file.txt > `> one > two > three >`

- The use of **iterators** pervades and unifies Python.
- Behind the scenes, the `for` statement calls `iter()` on the container object.
- The function returns an iterator object that defines the method `__next__()` which accesses elements in the container one at a time.
- When there are no more elements, `__next__()` raises a `StopIteration` exception which tells the `for` loop to terminate.
- You can call the `__next__()` method using the `next()` built-in function; this example shows how it all works:

[182]:
```
mylist = ["apple", "orange", "coconut"]
itr = iter(mylist)
itr
print(itr)

item = next(itr)
print(item)
item = next(itr)
print(item)
item = next(itr)
print(item)
```

```
item = next(itr)
print(item)
```

```
<list_iterator object at 0x7ff4f0625c40>
apple
orange
coconut


       ␣
 ↪---------------------------------------------------------------------------

        StopIteration                             Traceback (most recent call␣
 ↪last)

        <ipython-input-182-be91d8882743> in <module>
         10 item = next(itr)
         11 print(item)
    ---> 12 item = next(itr)
         13 print(item)


        StopIteration:
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. - Define an `__iter__()` method which returns an object with a `__next__()` method. - If the class defines `__next__()`, then `__iter__()` can just return self:

[188]:
```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

rev = Reverse("hello world")
iter(rev)

for char in rev:
    print(char, end='')
```

```
dlrow olleh
```

1.9 Generators

- 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 **next()** is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed).
- An example shows that generators can be trivially easy to create:

[189]:
```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

for char in reverse("hello world"):
    print(char, end='')
```

```
dlrow olleh
```

- Anything that can be done with generators can also be done with class-based iterators as described in the previous section. 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. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.
- 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.

1.10 Generator Expressions

- Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets.
- These expressions are designed for situations where the generator is used right away by an enclosing function.
- Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

[199]:
```
sq = sum(i*i for i in range(10))                   # sum of squares
print(sq)

xvec = [10, 20, 30]
yvec = [7, 5, 3]
dotprod = sum(x*y for x,y in zip(xvec, yvec))         # dot product
print(dotprod)
```

```
page = ["hello world","the world is great", "great morning"]
unique_words = set(word for line in page for word in line.split())
print(unique_words)

class Grad:
    def __init__(self, name, gpa):
        self.gpa = gpa
        self.name = name

graduates = [Grad("Ali", 58), Grad("Tan", 87), Grad("Chong", 67) ]
valedictorian = max((student.gpa, student.name) for student in graduates)
print(valedictorian)

data = 'golf'
mylist = list(data[i] for i in range(len(data)-1, -1, -1))
print(mylist)
```

```
285
260
{'world', 'the', 'hello', 'morning', 'great', 'is'}
(87, 'Tan')
['f', 'l', 'o', 'g']
```

1.11 When to use yield instead of return in Python?

https://www.geeksforgeeks.org/use-yield-keyword-instead-return-keyword-python/

The yield statement suspends function's execution and sends a value back to the caller, but retains enough state to enable function to resume where it is left off. When resumed, the function continues execution immediately after the last yield run. This allows its code to produce a series of values over time, rather than computing them at once and sending them back like a list.

[200]:
```
# A Simple Python program to demonstrate working
# of yield

# A generator function that yields 1 for the first time,
# 2 second time and 3 third time
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

# Driver code to check above generator function
for value in simpleGeneratorFun():
    print(value)
```

```
1
2
3
```

`return` sends a specified value back to its caller whereas **`yield`** can produce a sequence of values. We should use yield **when we want to iterate over a sequence, but don't want to store the entire sequence in memory**.

Yield are **used in Python generators**. A generator function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. **If the body of a def contains yield, the function automatically becomes a generator function**.

[202]:
```python
# A Python program to generate squares from 1
# to 100 using yield and therefore generator

# An infinite generator function that prints
# next square number. It starts with 1
def nextSquare():
    i = 1;

    # An Infinite loop to generate squares
    while True:
        yield i*i
        i += 1  # Next execution resumes
                # from this point

# Driver code to test above generator
# function
for num in nextSquare():
    if num > 100:
         break
    print(num)
```

```
1
4
9
16
25
36
49
64
81
100
```

2 END OF The Python Tutorial -> Classes