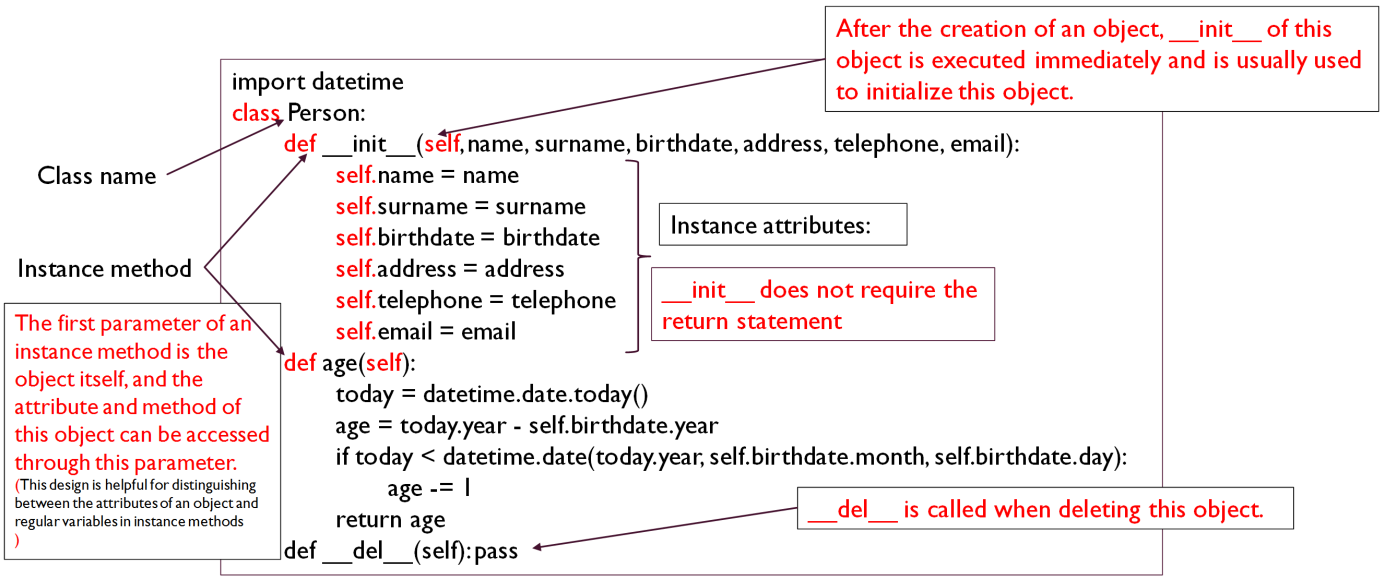
Classes

Classes are used to group related data and functions together.



```
import datetime
class Person:
    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate
        self.address = address
        self.telephone = telephone
        self.email = email
    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year
        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1
        return age
    def __del__(self):pass
```

After the creation of an object, __init__ of this object is executed immediately and is usually used to initialize this object.

Class name

Instance method

Instance attributes:

__init__ does not require the return statement

The first parameter of an instance method is the object itself, and the attribute and method of this object can be accessed through this parameter. (This design is helpful for distinguishing between the attributes of an object and regular variables in instance methods)

__del__ is called when deleting this object.

In [2]:

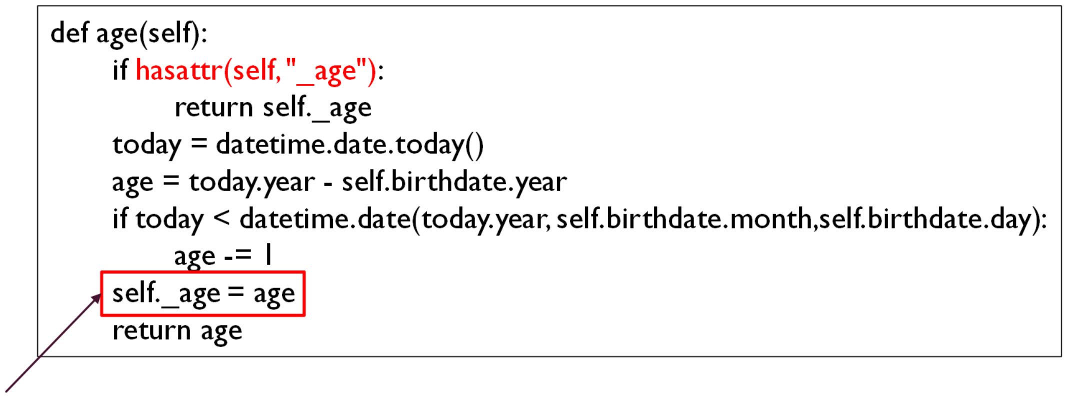
```python
import datetime
class Person:
    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate
        self.address = address
        self.telephone = telephone
        self.email = email
    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year
        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1
        return age
    def __del__(self): pass
```

In [3]:

```python
person = Person(
"Jane",
"Doe",
datetime.date(1992, 3, 12), # year, month, day
"No. 12 Short Street, Greenville",
"555 456 0987",
"jane.doe@example.com"
)
print(person.name)
print(person.email)
print(person.age())
```

```
Jane
jane.doe@example.com
26
```

Instance attributes can also be defined in other instance functions.



```
def age(self):
    if hasattr(self, "_age"):
        return self._age
    today = datetime.date.today()
    age = today.year - self.birthdate.year
    if today < datetime.date(today.year, self.birthdate.month,self.birthdate.day):
        age -= 1
    self._age = age
    return age
```

In Python, an attribute or method name starting with _ is often a private internal property and should not be accessed directly.

Instance attributes can also be defined outside the class definition

In [ ]:

```python
person.pets = ['cat', 'cat', 'dog']
```

getattr, setattr, delattr, AND hasattr

- getattr(object, attribute_name, default_value) can be used to retrieve the attribute value of an object.
- setattr(object, attribute_name, new_value) can be used to set the value of an attribute.
- hasattr(object, attribute_name) can be used to check if this object has an attribute.
- delattr(object, attribute_name) can be used to remove an attribute from an object.
- If getattr and delattr inquire non-existing attributes, an AttributeError exception arises.

In [4]:

```
class person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
a = person("harry", "potter")
```

In [5]:

```
print(getattr(a,"first_name", None))
```

```
harry
```

In [7]:

```
setattr(a, "age", 20)
print(a.age) # 20
```

```
20
```

In [8]:

```
print(hasattr(a,"first_name")) # True
```

```
True
```

@property, @property_name.setter, AND @property_name.deleter

- Some properties are derived from existing attributes, and getter methods can be implemented for accessing these properties like accessing them directly.
- The setter methods are used to update the values of those properties.
- The delete methods are used to delete those properties.
- The getter, setter and deleter methods must all have the same name!

without setter

```
class Square:
    def __init__(self,width):
        self.width = width
    def area(self):
        return self.width**2

a = Square(10)
print(a.area())
```

In [9]:

```
# with setter
class Square:
    def __init__(self,width):
        self.width = width
    @property
    def area(self):
        return self.width**2
    @area.setter
    def area(self, value):
        self.width = value**0.5
    @area.deleter
    def area(self):
        del self.width
```

In [13]:

```
a = Square(10)
print(a.area)
a.area = 16
print(a.width)
```

```
100
4.0
```

Inspecting an object

The dir built-in function can be used to list the properties of an object or a class.

```
In [14]:
```

```python
dir(a)
```

```
Out[14]:
```

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

In [15]:

```python
del a.area
dir(a)
```

Out[15]:

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

In [12]:

```
dir(Square)
```

Out[12]:

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

- __init__: the initialization method of an object, which is called when the object is created.
- __str__: the string representation method of an object, which is called when you use the str function to convert that object to a string. (str(x) is equivalent to x.__str)
- __repr__: the internal representation of an object, which is called when you use the repr function to show the object. For the internal object x, eval(repr(x)) often creates an object equal to x. (repr(x) is equivalent to x.__repr__())
- __class__: an attribute which stores the class (or type) of an object – this is what is returned when you use the type function on the object.
- __eq__: a method which determines whether this object is equal to another. There are also other methods for determining if it's not equal, less than, etc.. These methods are used in object comparisons, for example when we use the equality operator == to check if two objects are equal.
- __add__ is a method which allows this object to be added to another object. There are equivalent methods for all the other arithmetic operators. Not all objects support all arithmetic operations – numbers have all of these methods defined, but other objects may only have a subset.
- __iter__: a method which returns an iterator over the object – we will find it on strings, lists and other iterables. It is executed when we use the iter function on the object.
- __len__: a method which calculates the length of an object – we will find it on sequences. It is executed when we use the len function of an object.
- __dict__: a dictionary which contains all the instance attributes of an object, with their names as keys. It can be useful if we want to iterate over all the attributes of an object. __dict__ does not include any methods, class attributes or special default attributes like __class__. (vars(x) is equivalent to x.__dict__)

Magic Methods

Magic Methods

Operator	Method	Operator	Method	Operator	Method
<	object.__lt__(self,other)	-	object.__neg__(self)	+=	object.__iadd__(self,other)
<=	object.__le__(self,other)	+	object.__pos__(self)	-=	object.__isub__(self,other)
==	object.__eq__(self,other)	abs()	object.__abs__(self)	*=	object.__imul__(self,other)
!=	object.__ne__(self,other)	~	object.__invert__(self)	/=	object.__idiv__(self,other)
>=	object.__ge__(self,other)	complex	object.__complex__(self)	//=	object.__ifloordiv__(self,other)
>	object.__gt__(self,other)	int()	object.__int__(self)	%=	object.__imod__(self,other)
		long()	object.__long__(self)	**=	object.__ipow__(self,other)
		float()	object.__float__(self)	<<=	object.__ilshift__(self,other)
		oct()	object.__oct__(self)	>>=	object.__irshift__(self,other)
		hex()	object.__hex__(self)	&=	object.__iand__(self,other)
				^=	object.__ixor__(self,other)
				|=	object.__ior__(self,other)

Magic methods used with the with operator[ ]

Operator	Method
obj[key]	object.__getitem__(self,key)
obj[key]=value	object.__setitem__(self,key,value)
del obj[key]	object.__delitem__(self,key)

Magic methods used with the with statement:

Operator	Method
Enter	object.__enter__(self)
Exit	object.__exit__(self, type, value, trace)

Overriding Magic Methods

In [17]:

```python
class FractionalNumber:
    def __init__(self, numer, denom):
        self.n = numer
        self.d = denom
    def __add__(self, other):
        return FractionalNumber(self.d*other.n+self.n*other.d,self.d*other.d)
    def __sub__(self, other):
        return FractionalNumber(self.d*other.n-self.n*other.d,self.d*other.d)
    def __mul__(self,other):
        return FractionalNumber(self.n*other.n,self.d*other.d)
    def __truediv__(self,other):
        return FractionalNumber(self.d*other.d,self.n*other.n)
    def __str__(self):
        return str(self.n)+"/"+str(self.d)
    def __repr__(self):
# Tell Python how to represent FractionalNumber.
# The output string must be able to be evaluated by eval.
        return "FractionalNumber({},{})".format(self.n,self.d)
```

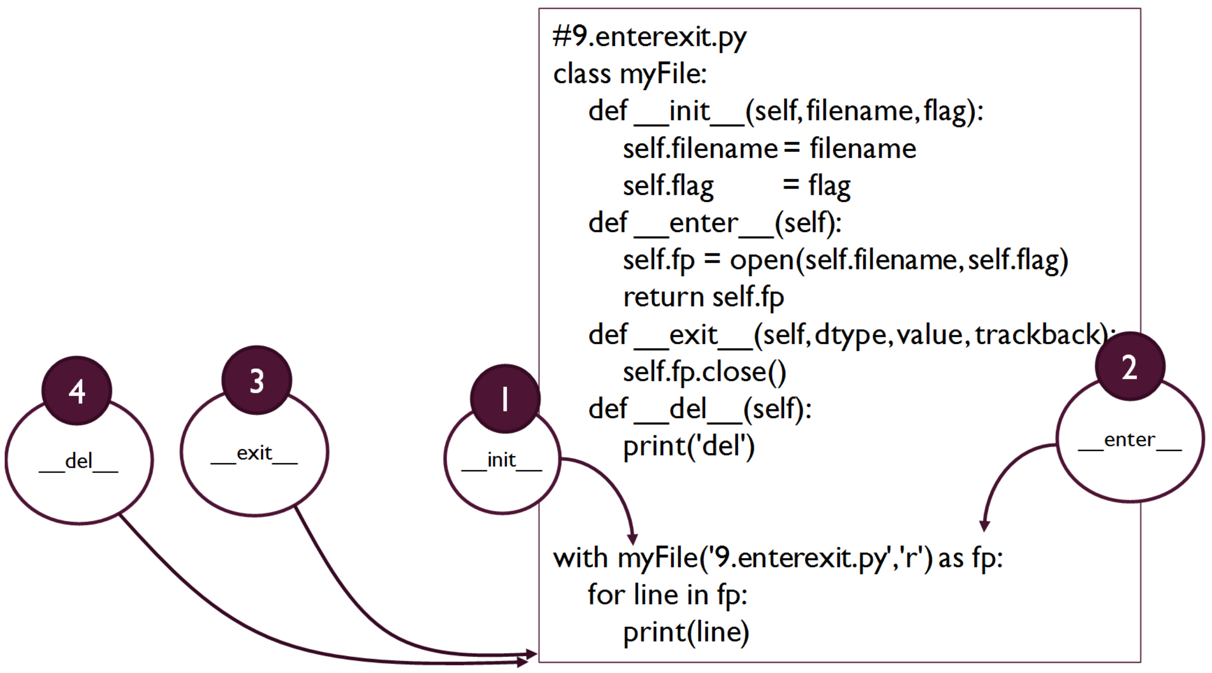
In [23]:

```python
a=FractionalNumber(1,2)
b=FractionalNumber(1,3)
c=a+b
print(c,eval('c'))
```

```
5/6 5/6
```

The enter and exit magic methods are used to implement objects which can be used with the with statement.

- **enter** is called when entering the with statement
- **exit** is called when leaving the with statement



```python
#9.enterexit.py
class myFile:
    def __init__(self,filename,flag):
        self.filename = filename
        self.flag       = flag
    def __enter__(self):
        self.fp = open(self.filename,self.flag)
        return self.fp
    def __exit__(self,dtype,value,trackback):
        self.fp.close()
    def __del__(self):
        print('del')

with myFile('9.enterexit.py','r') as fp:
    for line in fp:
        print(line)
```

In [25]:

```
class myFile:
    def __init__(self):
        print('init')

    def __enter__(self):
        print('enter')
        return 1
    def __exit__(self, dtype, value, trackback):
        print('exit')

    def __del__(self):
        print('del')


with myFile() as fp:
    print('hello')
```

```
init
enter
hello
exit
del
```

__iter__() of an object x is a method which returns an iterator over the object. iter(x) in fact calls x.__iter__().

In [27]:

```
class myLinkedList:
    def __init__(self,num):
        self.head = None
        for i in range(num):
            self.insert_head(i)

    def insert_head(self,data):
        self.head = (data,self.head)

    def __del__(self):
        p = self.head
        while p is not None:
            q = p
            p = p[1]
            del q

    def __iter__(self):
        self.iter_p = self.head
        return self

    def __next__(self):
        if self.iter_p is None:
            raise StopIteration
        p = self.iter_p
        self.iter_p = p[1]
        return p[0]
```

In [31]:

```
d = myLinkedList(5)
for p in d:
    print(p)
```

```
4
3
2
1
0
```

In [33]:

```
p = iter(d)
try:
    while True:
        v = next(p)
        print(v)
except StopIteration:
    pass
```

```
4
3
2
1
0
```

Class Attributes

- If all instances of a class type share some properties, these properties can be defined as class attributes or methods.

```
class Person:
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms') # class attribute
    def __init__(self, title, name, surname):
        if title not in self.TITLES:
            raise ValueError("%s is not a valid title." % title)
        self.title = title
        self.name = name
        self.surname = surname
```

- Class attributes are shared by all instances.
- Class attributes can be accessed through self.class_attribute or class_name.class_attribute, e.g., self.TITLES or Person.TITLES.
- When an instance attribute with the same name as a class attribute is defined, this instance attribute overrides the class attribute.
- When a mutable class attribute is modified in-place, other instances of this class may be affected.
- Class attributions can be used as variables in method definitions.

In [42]:

```
class Person:
    TITLES = ['Dr', 'Mr', 'Mrs', 'Ms'] # class attribute
    def __init__(self, title, name, surname, allowed_titles=TITLES):
        if title not in allowed_titles:
            raise ValueError("%s is not a valid title." % title)
        self.title = title
        self.name = name
        self.surname = surname
    def change_my_title(self):
        self.TITLES = ['Mr'] # the instance attribute overrides the class attribute
    def change_class_title(self):
        Person.TITLES = ['Mrs'] # the class attribute is modified in-place
```

In [43]:

```
a = Person('Dr', 'Roy', 'Davies')
b = Person('Dr', 'Linda', 'Shapiro')
print(a.TITLES,b.TITLES,Person.TITLES, Person.TITLES is a.TITLES, Person.TITLES is b.TITLES
a.change_my_title() # a's instance attribute TITLES overrides the class attribute
print(a.TITLES,b.TITLES,Person.TITLES, Person.TITLES is a.TITLES, Person.TITLES is b.TITLES
a.change_class_title() # the class attribute is modified in-place
print(a.TITLES,b.TITLES,Person.TITLES, Person.TITLES is a.TITLES, Person.TITLES is b.TITLES
```

```
['Dr', 'Mr', 'Mrs', 'Ms'] ['Dr', 'Mr', 'Mrs', 'Ms'] ['Dr', 'Mr', 'Mrs', 'M
s'] True True
['Mr'] ['Dr', 'Mr', 'Mrs', 'Ms'] ['Dr', 'Mr', 'Mrs', 'Ms'] False True
['Mr'] ['Mrs'] ['Mrs'] False True
```

Class Decorators

The @classmethod decorator can be used to define class methods, which are shared between all instances.

```
class Person:
    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
    #…
    @classmethod
    def from_text_file(cls, filename):
        #extract all the parameters from the text file
        return cls( *params) # this is the same as calling Person( *params)
```

The subclass of Person also inherits from_text_file and subclass. from_text_file returns the instance of subclass.

In [47]:

```
class my_dict(dict):
    pass
print(type(dict.fromkeys([1,2,3]))) # dict
print(type(my_dict.fromkeys([1,2,3]))) # __main__.my_dict
```

```
<class 'dict'>
<class '__main__.my_dict'>
```

The @staticmethod decorator can be used to define a static method, which does not have the calling object or class name passed into it as the first parameter.

In [48]:

```
class Person:
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
    def fullname(self): # instance method
        # instance object accessible through self
        return "%s %s" % (self.name, self.surname)
    @classmethod
    def allowed_titles_starting_with(cls, startswith): # class method
        # class or instance object accessible through cls
        return [t for t in cls.TITLES if t.startswith(startswith)]
    @staticmethod
    def allowed_titles_ending_with(endswith): # static method
        # no parameter for class or instance object
        # we have to use Person directly
        return [t for t in Person.TITLES if t.endswith(endswith)]
jane = Person("Jane", "Smith")
print(jane.fullname())
print(jane.allowed_titles_starting_with("M"))
print(Person.allowed_titles_starting_with("M"))
print(jane.allowed_titles_ending_with("s"))
print(Person.allowed_titles_ending_with("s"))
```

```
Jane Smith
['Mr', 'Mrs', 'Ms']
['Mr', 'Mrs', 'Ms']
['Mrs', 'Ms']
['Mrs', 'Ms']
```

In [ ]:

```

```