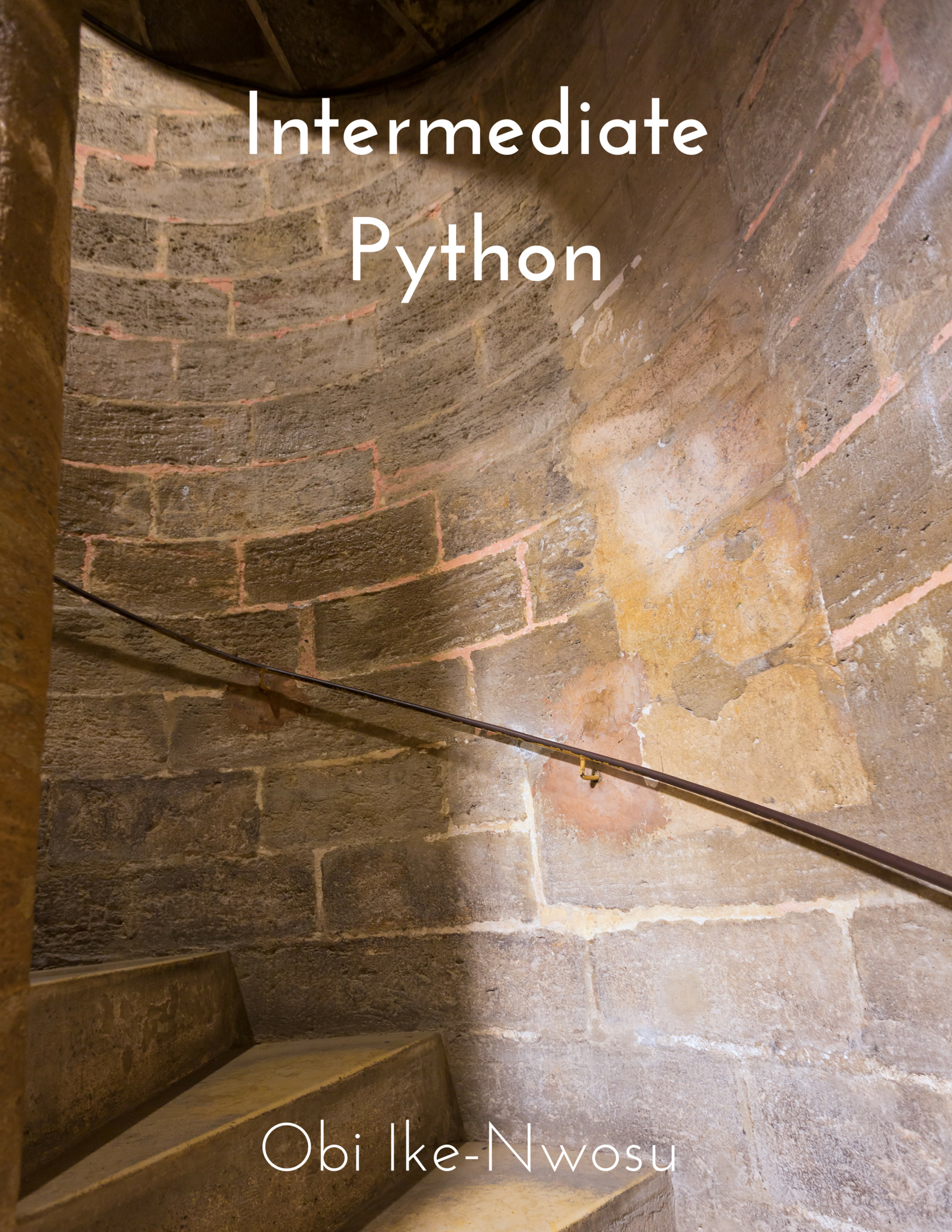
Intermediate
Python
Obi Ike-Nwosu

# Intermediate Python

Obi Ike-Nwosu

This book is for sale at http://leanpub.com/intermediatepython

This version was published on 2016-11-27

# Contents

# 1. Intermezzo: Glossary

A number of terms and esoteric python functions are used throughout this book and a good understanding of these terms is integral to gaining a better. and deeper understanding of python. A description of these terms and functions is provided in the sections that follow.

## 1.1 Names and Binding

In python, objects are referenced by *names*. `names` are analogous to variables in `C++` and `Java`.

```
>>> x = 5
```

In the above, example, `x` is a name that references the object, `5`. The process of *assigning* a reference to `5` to `x` is called *binding*. A binding causes a name to be associated with an object in the innermost scope of the currently executing program. Bindings may occur during a number of instances such as during variable assignment or function or method call when the supplied parameter is bound to the argument. It is important to note that names are just symbols and they have no *type* associated with them; **names are just references to objects that actually have types**

## 1.2 Code Blocks

A code block is a piece of program code that is executed as a single unit in python. Modules, functions and classes are all examples of code blocks. Commands typed in interactively at the REPL, script commands run with the `-c` option are also code blocks. A code block has a number of name-spaces associated with it. For example, a module code block has access to the `global`name-space while a function code block has access to the `local` as well as the `global` name-spaces.

## 1.3 Name-spaces

A *name-space* as the name implies is a context in which a given set of names is bound to objects. Name-spaces in python are currently implemented as dictionary mappings. The *built-in* name-space is an example of a name-space that contains all the built-in functions and this can be accessed by entering `__builtins__.__dict__` at the terminal (the result is of a considerable amount). The interpreter has access to multiple name-spaces including *the global name-space*, *the built-in name-space* and *the local name-space*. These name-spaces are created at different times and have different lifetimes. For example, a new *local* name-space is created at the invocation of a function and

forgotten when the function exits or returns. The *global* name-space is created at the start of the execution of a module and all names defined in this name-space are available module-wide while the *built-in* name-space comes into existence when the interpeter is invoked and contains all the built-in names. These three name-spaces are the main name-space available to the interpreter.

## 1.4 Scopes

A scope is an area of a program in which a set of name bindings (name-spaces) is visible and directly accessible. Direct access is an important characteristic of a scope as will be explained when classes are discussed. This simply means that a name, `name`, can be used as is, without the need for dot notation such as `SomeClassOrModule.name` to access it. At runtime, the following scopes may be available.

1. Inner most scope with local names
2. The scope of enclosing functions if any (this is applicable for nested functions)
3. The current module's globals scope
4. The scope containing the builtin name-space.

Whan a name is used in python, the interpreter searches the name-spaces of the scopes in ascending order as listed above and if the name is not found in any of the name-spaces, an exception is raised. Python supports static scoping also known as lexical scoping; this means that the visibility of a set of name bindings can be inferred by only inspecting the program text.

### Note

Python has a quirky scoping rule that prevents a reference to an object in the *global* scope from being modified in a local scope; such an attempt will throw an `UnboundLocalError` exception. In order to modify an object from the global scope within a local scope, the `global` keyword has to be used with the object name before modification is attempted. The following example illustrates this.

```
>>> a = 1
>>> def inc_a(): a += 2
...
>>> inc_a()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "<stdin>", line 1, in inc_a
UnboundLocalError: local variable 'a' referenced before assignment
```

In order to modify the object from the global scope, the `global` statement is used as shown in the following snippet.

```
>>> a = 1
>>> def inc_a():
...     global a
...     a += 1
...
>>> inc_a()
>>> a
2
```

Python also has the `nonlocal` keyword that is used when there is a need to modify a variable bound in an outer non-global scope from an inner scope. This proves very handy when working with nested functions (also referred to as closures). A very trivial illustration of the `nonlocal` keyword in action is shown in the following snippet that defines a simple counter object that counts in ascending order.

```
>>> def make_counter():
...     count = 0
...     def counter():
...         nonlocal count # nonlocal captures the count binding from enclosing scope not global\
 scope
...         count += 1
...         return count
...     return counter
...
>>> counter_1 = make_counter()
>>> counter_2 = make_counter()
>>> counter_1()
1
>>> counter_1()
2
>>> counter_2()
1
>>> counter_2()
2
```

## 1.5 `eval()`

`eval` is a python built-in method for dynamically executing python expressions in a string (the content of the string must be a valid python expression) or code objects. The function has the following signature `eval(expression, globals=None, locals=None)`. If supplied, the `globals` argument to the `eval` function must be a dictionary while the `locals` argument can be any mapping. The evaluation of the supplied expression is done using the *globals* and *locals* dictionaries as the global and local name-spaces. If the `__builtins__` is absent from the globals dictionary, the current globals are copied into globals before expression is parsed. This means that the expression will have either full or restricted access to the standard built-ins depending on the execution environment; this way the exection environment of `eval` can be restricted or *sandboxed*. `eval` when called returns the result of executing the expression or code object for example:

```python
>>> eval("2 + 1") # note the expression is in a string
3
```

Since `eval` can take arbitrary code obects as argument and return the value of executing such expressions, it along with `exec`, is used in executing arbitrary Python code that has been compiled into code objects using the `compile` method. Online Python interpreters are able to execute python code supplied by their users using both `eval` and `exec` among other methods.

## 1.6 exec()

`exec` is the counterpart to `eval`. This executes a string interpreted as a suite of python statements or a code object. The code supplied is supposed to be valid as file input in both cases. `exec` has the following signature: `exec(object[, globals[, locals]])`. The following is an example of `exec` using a string and the current name-spaces.

```
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct  5 2014, 20:42:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
# the acct.py file is located somewhere on file
>>> cont = open('acct.py', 'r').read()
>>> cont
'class Account:\n    """base class for representing user accounts"""\n    num_accounts = 0\n\n    de\
f __init__(self, name, balance):\n        self.name = name \n        self.balance = balance \n      \
  Account.num_accounts += 1\n\n    def del_account(self):\n        Account.num_accounts -= 1\n\n    \
def __getattr__(self, name):\n        """handle attribute reference for non-existent attribute"""\n \
       return "Hey I dont see any attribute called {}".format(name)\n\n    def deposit(self, amt):\n\
        self.balance = self.balance + amt \n\n    def withdraw(self, amt):\n        self.balance = s\
elf.balance - amt \n\n    def inquiry(self):\n        return "Name={}, balance={}".format(self.name,\
 self.balance) \n\n'
>>> exec(cont)
# exec content of file using the default name-spaces
>>> Account # we can now reference the account class
<class '__main__.Account'>
>>>
```

In all instances, if optional arguments are omitted, the code is executed in the current scope. If only the `globals` argument is provided, it has to be a dictionary, that is used for both the global and the local variables. If `globals` and `locals` are given, they are used for the global and local variables, respectively. If provided, the `locals` argument can be any mapping object. If the `globals` dictionary does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key. One can control the `builtins` that are available to the executed code by inserting custom `__builtins__` dictionary into `globals` before passing it to `exec()` thus creating a sandbox.

# 2. Object Oriented Programming

Classes are the basis of object oriented programming in python and are one of the basic organizational units in a python program.

## 2.1 The Mechanics of Class Definitions

The `class` statement is used to define a new type. The class statement defines a set of attributes, variables and methods, that are associated with and shared by a collection of instances of such a class. A simple class definition is given below:

```
class Account(object):
    num_accounts = 0

    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
        Account.num_accounts += 1

    def del_account(self):
        Account.num_accounts -= 1

    def deposit(self, amt):
        self.balance = self.balance + amt

    def withdraw(self, amt):
        self.balance = self.balance - amt

    def inquiry(self):
        return self.balance
```

Class definitions introduce class objects, instance objects and method objects.

### Class Objects

The execution of a class statement creates a class object. At the start of the execution of a class statement, a new *name-space* is created and this serves as the name-space into which all class attributes go; unlike languages like Java, this name-space does not create a new local scope that can be used by class methods hence the need for fully qualified names when accessing attributes. The `Account` class from the previous section illustrates this; a method trying to access the `num_accounts` variable must use the fully qualified name, `Account.num_accounts` else an error results such as when the fully qualified name is not used in the `__init__` method as shown below:

```
class Account(object):
    num_accounts = 0

    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
        num_accounts += 1

    def del_account(self):
        Account.num_accounts -= 1

    def deposit(self, amt):
        self.balance = self.balance + amt

    def withdraw(self, amt):
        self.balance = self.balance - amt

    def inquiry(self):
        return self.balance

>>> acct = Account('obi', 10)
Traceback (most recent call last):
  File "python", line 1, in <module>
  File "python", line 9, in __init__
UnboundLocalError: local variable 'num_accounts' referenced before assignment
```

At the end of the execution of a class statement, a class object is created; the scope preceding the class definition is reinstated, and the class object is bound in this scope to the class name given in the class definition header.

A little diversion here. One may ask, *if the class created is an object then what is the class of the class object?*. In accordance with the Python philosophy that *every value is an object* , the class object does indeed have a class which it is created from; this is the `type` class.

```
>>> type(Account)
<class 'type'>
```

So just to confuse you a bit, the type of a type, *the Account type*, is `type`. To get a better understanding of the fact that a class is indeed an object with its own class we go behind the scenes to explain what really goes on during the execution of a class statement using the `Account` example from above.

```
>>>class_name = "Account"
>>>class_parents = (object,)
>>>class_body = """
num_accounts = 0

def __init__(self, name, balance):
    self.name = name
    self.balance = balance
    num_accounts += 1

def del_account(self):
    Account.num_accounts -= 1

def deposit(self, amt):
    self.balance = self.balance + amt

def withdraw(self, amt):
    self.balance = self.balance - amt

def inquiry(self):
    return self.balance
"""
# a new dict is used as local name-space
>>>class_dict = {}

#the body of the class is executed using dict from above as local
# name-space
>>>exec(class_body, globals(), class_dict)

# viewing the class dict reveals the name bindings from class body
>>> class_dict
{'del_account': <function del_account at 0x106be60c8>, 'num_accounts': 0, 'inquiry': <function i\
nquiry at 0x106beac80>, 'deposit': <function deposit at 0x106be66e0>, 'withdraw': <function withdraw\
 at 0x106be6de8>, '__init__': <function __init__ at 0x106be2c08>}

# final step of class creation
>>>Foo = type(class_name, class_parents, class_dict)
# view created class object
>>>Account
<class '__main__.Account'>
>>>type(Account)
<type 'type'>
```

During the execution of class statement, the interpreter *kind of* carries out the following steps behind the scene (greatly simplified here):

1. The body of the class statement is isolated into a code object.
2. A class dictionary representing the name-space for the class is created.
3. The code object representing the body of the class is executed within this name-space.

4. During the final step, the class object is created by instantiating the `type` class, passing in the class name, base classes and class dictionary as arguments. The `type` class used here in creating the `Account` class object is a **meta-class**, the class of a class. The meta-class used in the class object creation can be explicitly specified by supplying the `metaclass` keyword argument in the `class` definition. In the case that this is not supplied, the base classes if any are checked for a meta-class. If no base classes are supplied, then the default `type()` metaclass is used. More about meta-classes is discussed in subsequent chapters.

Class objects support *attribute reference* and *object instantiation.* Attributes are referenced using the standard dot syntax; an object followed by dot and then attribute name: `obj.name`. Valid attribute names are all the variable and method names present in the class' name-space when the class object was created. For example:

```
>>> Account.num_accounts
0
>>> Account.deposit
>>> <unbound method Account.deposit>
```

Object instantiation is carried out by calling the class object like a normal function with required parameters for the `__init__` method of the class as shown in the following example:

```
>>> Account("obi", 0)
```

An instance object that has been initialized with supplied arguments is returned from instantiation of a class object. In the case of the `Account` class, the account name and account balance are set and, the number of instances is incremented by `1` in the `__init__` method.

## Instance Objects

If class objects are the cookie cutters then instance objects are the cookies that are the result of instantiating class objects. Instance objects are returned after the correct initialization of a class just as shown in the previous section. Attribute references are the only operations that are valid on instance objects. Instance attributes are either data attribute, better known as instance variables in languages like Java, or method attributes.

## Method Objects

If `x` is an instance of the `Account` class, `x.deposit` is an example of a method object. Method objects are similar to functions however during a method definition, an extra argument is included in the arguments list, the `self` argument. This `self` argument refers to an instance of the class but *why do we have to pass an instance as an argument to a method?* This is best illustrated by a method call such as the following.

```
>>> x = Account()
>>> x.inquiry()
10
```

**But what exactly happens when an instance method is called?** It can be observed that `x.inquiry()` is called without an argument above, even though the method definition for `inquiry()` requires the `self` argument. *What happened to this argument?*

In the example from above, the call to `x.inquiry()` is exactly equivalent to `Account.inquiry(x)`; notice that the object instance, `x`, is being passed as argument to the method - this is the `self` argument. Invoking an object method with an argument list is equivalent to invoking the corresponding method from the object's class with an argument list that is created by inserting the method's object at the start of the list of argument. In order to understand this, note that methods are stored as functions in class `dicts`.

```
>>> type(Account.inquiry)
<class 'function'>
```

To fully understand how this transformation takes place one has to understand descriptors and Python's attribute references algorithm. These are discussed in subsequent sections of this chapter. In summary, the method object is a wrapper around a function object; when the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the underlying function object is called with this new argument list. This applies to all instance method objects including the `__init__` method. Note that the `self` argument is actually not a keyword; the name, `self` is just a convention and any valid argument name can be used as shown in the `Account` class definition below.

```
class Account(object):
    num_accounts = 0

    def __init__(obj, name, balance):
        obj.name = name
        obj.balance = balance
        Account.num_accounts += 1

    def del_account(obj):
        Account.num_accounts -= 1

    def deposit(obj, amt):
        obj.balance = obj.balance + amt

    def withdraw(obj, amt):
        obj.balance = obj.balance - amt

    def inquiry(obj):
        return obj.balance
```

```
>>> Account.num_accounts
0
>>> x = Account('obi', 0)
>>> x.deposit(10)
>>> Account.inquiry(x)
10
```

# 2.2 Customizing User-defined Types

Python is a very flexible language providing user with the ability to customize classes in ways that are unimaginable in other languages. Attribute access, class creation and object initialization are a few examples of ways in which classes can be customized. User defined types can also be customized to behave like built-in types and support special operators and syntax such as `*`, `+`, `-`, `[]` etc.

All these customization is possible because of methods that are called *special* or *magic* methods. Python special methods are just ordinary python methods with double underscores as prefix and suffix to the method names. Special methods have already encountered in this book. An example is the `__init__` method that is called to initialize class instances; another is the `__getitem__` method invoked by the index, `[]` operator; an index such as `a[i]` is translated by the interpreter to a call to `type(a).__getitem__(a, i)`. Methods with the double underscore as prefix and suffix are just ordinary python methods; users can define their own class methods with method names prefixed and suffixed with the double underscore and use it just like normal python methods. This is however not the conventional approach to defining normal user methods.

User defined classes can also implement these special methods; a corollary of this is that built-in operators such as `+` or `[]` can be adapted for use by user defined classes. This is one of the essence of *polymorphism* in Python. In this book, special methods are grouped according to the functions they serve. These groups include:

## Special methods for instance creation

The `__new__` and `__init__` special methods are the two methods that are integral to instance creation. New class instances are created in a two step process; first the static method, `__new__`, is called to create and return a new class instance then the `__init__` method is called to to initialize the newly created object with supplied arguments. A very important instance in which there is a need to override the `__new__` method is when sub-classing built-in immutable types. Any initialization that is done in the sub-class must be done before object creation. This is because once an immutable object is created, its value cannot be changed so it makes no sense trying to carry out any function that modifies the created object in an `__init__` method. An example of sub-classing is shown in the following snippet in which whatever value is supplied is rounded up to the next integer.

```
>>> import math
>>> class NextInteger(int):
...     def __new__(cls, val):
...         return int.__new__(cls, math.ceil(val))
...
>>> NextInteger(2.2)
3
>>>
```

Attempting to do the `math.ceil` operation in an `__init__` method will cause the object initialization to fail. The `__new__` method can also be overridden to create a `Singleton` super class; subclasses of this class can only ever have a single instance throughout the execution of a program; the following example illustrates this.

```
class Singleton:

    def __new__(cls, *args, **kwds):
        it = cls.__dict__.get("__it__")
        if it is None:
            return it
        cls.__it__ = it = object.__new__(cls)
        it.init(*args, **kwds)
        return it

    def __init__(self, *args, **kwsds):
        pass
```

It is worth noting that when implementing the `__new__` method, the implementation must call its base class' `__new__` and the implementation method must return an object.

Users are already familiar with defining the `__init__` method; the `__init__` method is overridden to perform attribute initialization for an instance of a mutable types.

## Special methods for attribute access

The special methods in this category provide means for customizing attribute references; this maybe in order to access or set such an attribute. This set of special methods available for this include:

1. `__getattr__`: This method can be implemented to handle situations in which a referenced attribute cannot be found. This method is ***only*** called when an attribute that is referenced is neither an instance attribute nor is it found in the class tree of that object. This method should return some value for the attribute or raise an `AttributeError` exception. For example, if $x$ is an instance of the `Account` class defined above, trying to access an attribute that does not exist will result in a call to this method as shown in the following snippet

```
    class Account(object):
        num_accounts = 0

        def __init__(self, name, balance):
            self.name = name
            self.balance = balance
            Account.num_accounts += 1

        def del_account(self):
            Account.num_accounts -= 1

        def __getattr__(self, name):
            return "Hey I don't see any attribute called {}".format(name)

        def deposit(self, amt):
            self.balance = self.balance + amt

        def withdraw(self, amt):
            self.balance = self.balance - amt

        def inquiry(self):
            return "Name={}, balance={}".format(self.name, self.balance)

    >>> x = Account('obi', 0)
    >>> x.balaance
    Hey I dont see any attribute called balaance
```

Care should be taken with the implementation of `__getattr__` because if the implementation references an instance attribute that does not exist, an infinite loop may occur because the `__getattr__` method is called successively without end.

```
    class Account(object):
        num_accounts = 0

        def __init__(self, name, balance):
            self.name = name
            self.balance = balance
            Account.num_accounts += 1

        def del_account(self):
            Account.num_accounts -= 1

        def __getattr__(self, name):
            return self.namee # trying to acess a variable that doesnt exist will result in __getatt\
r___ calling itself over and over again

        def deposit(self, amt):
            self.balance = self.balance + amt

        def withdraw(self, amt):
```

```
            self.balance = self.balance - amt

        def inquiry(self):
            return "Name={}, balance={}".format(self.name, self.balance)

    >>> x = Account('obi', 0)
    >>> x.balaance # this will result in a RuntimeError: maximum recursion depth exceeded while call\
ing a Python object exception
```

1. `__getattribute__`: This method is implemented to customize the attribute access for a class. This method is **always** called unconditionally during attribute access for instances of a class.
2. `__setattr__`: This method is implemented to unconditionally handle all attribute assignment. `__setattr__` should insert the value being assigned into the dictionary of the instance attributes rather than using `self.name=value` which results in an infinite recursive call. When `__setattr__()` is used for instance attribute assignment, the base class method with the same name should be called such as `super().__setattr__(self, name, value)`.
3. `__delattr__`: This is implemented to customize the process of deleting an instance of a class. it is invoked whenever `del obj` is called.
4. `__dir__`: This is implemented to customize the list of object attributes returned by a call to `dir(obj)`.

## Special methods for Type Emulation

Built-in types in python have special operators that work with them. For example, numeric types support the + operator for adding two numbers, numeric types also support the - operator for subtracting two numbers, sequence and mapping types support the `[]` operator for indexing values held. Sequence types even also have support for the + operator for concatenating such sequences. User defined classes can be customized to *behave* like these built-in types where it makes sense. This can be done by implementing the special methods that are invoked by the interpreter when these *special* operators are encountered. The special methods that provide these functionalities for emulating built-in types can be broadly grouped into one of the following:

### Numeric Type Special Methods

The following table shows some of the basic operators and the special methods invoked when these operators are encountered.

| Special Method | Operator Description |
|---|---|
| `a.__add__(self, b)` | binary addition, a + b |
| `a.__sub__(self, b)` | binary subtraction, a - b |
| `a.__mul__(self, b)` | binary multiplication, a * b |
| `a.__truediv__(self, b)` | division of a by b |
| `a.__floordiv__(self, b)` | truncating division of a by b |
| `a.__mod__(self, b)` | a modulo b |
| `a.__divmod__(self, b)` | returns a divided by b, a modulo b |
| `a.__pow__(self, b[, modulo])` | a raised to the b*th* power |

Python has the concept of reflected operations; this was covered in the section on the `NotImplemented` of previous chapter. The idea behind this concept is that if the left operand of a binary arithmetic operation does not support a required operation and returns `NotImplemented` then an attempt is made to call the corresponding reflected operation on the right operand provided the type of both operands differ. An example of this rarely used functionality is shown in the following trivial example for emphasis.

```
        class MyNumber(object):
            def __init__(self, x):
                self.x = x

            def __str__(self):
                return str(self.x)

        >>> 10 - MyNumber(9) # int type, 10, does not know how to subtract MyNumber type and MyNumbe\
r does not know how to handle the operation too
        Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
        TypeError: unsupported operand type(s) for -: 'int' and 'MyNumber'
```

In the next snippet the class implements the reflected special method and this reflected method is called by the interpreter.

```
        class MyFixedNumber(MyNumber):
            def __rsub__(self, other): # reflected operation implemented
                return MyNumber(other - self.val)

        >>> (10 - MyFixedNumber(9)).val
        1
```

The following special methods implement reflected binary arithmetic operations.

| Special Method | Operator Description |
|---|---|
| `a.__radd__(self, b)` | reflected binary addition, a + b |
| `a.__rsub__(self, b)` | reflected binary subtraction, a - b |
| `a.__rmul__(self, b)` | reflected binary multiplication, a * b |
| `a.__rtruediv__(self, b)` | reflected division of a by b |
| `a.__rfloordiv__(self, b)` | reflected truncating division of a by b |
| `a.__rmod__(self, b)` | reflected a modulo b |
| `a.__rdivmod__(self, b)` | reflected a divided by b, a modulo b |
| `a.__rpow__(self, b[, modulo])` | reflected a raised to the b*th* power |

Another set of operators that work with numeric types are the augmented assignment operators. An example of an augmented operation is shown in the following code snippet.

```
>>> val = 10
>>> val += 90
>>> val
100
>>>
```

A few of the special methods for implementing augmented arithmetic operations are listed in the following table.

| Special Method | Description |
|---|---|
| `a.__iadd__(self, b)` | a += b |
| `a.__isub__(self, b)` | a -= b |
| `a.__imul__(self, b)` | a *= b |
| `a.__itruediv__(self, b)` | a //= b |
| `a.__ifloordiv__(self, b)` | a /= b |
| `a.__imod__(self, b)` | a %= b |
| `a.__ipow__(self, b[, modulo])` | a **= b |

## Sequence and Mapping Types Special Methods

Sequence and mapping are often referred to as container types because they can hold references to other objects. User-defined classes can emulate container types to the extent that this makes sense if such classes implement the special methods listed in the following table.

| Special Method | Description |
|---|---|
| `__len__(obj)` | returns length of `obj`. This is invoked to implement the built-in function `len()`. An object that doesn't define a `__bool__()` method and whose `__len__()` method returns zero is considered to be false in a Boolean context. |

| Special Method | Description |
|---|---|
| `__getitem__(obj, key)` | fetches item, `obj[key]`. For sequence types, the keys should be integers or slice objects. If key is of an inappropriate type, `TypeError` may be raised; if the key has a value outside the set of indices for the sequence, `IndexError` should be raised. For mapping types, if key is absent from the container, `KeyError` should be raised. |
| `__setitem__(obj, key, value)` | Sets `obj[key] = value` |
| `__delitem__(obj, key)` | deletes `obj[key]`. Invoked by del `obj[key]` |
| `__contains__(obj, key)` | Returns true if `key` is contained in `obj` and false otherwise. Invoked by a call to `key in obj` |
| `__iter__(self)` | This method is called when an iterator is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container. Iterator objects also need to implement this method; they are required to return themselves. This is also used by the `for..in` construct. |

Sequence types such as lists support the addition (for concatenating lists) and multiplication operators (for creating copies), `+ and *` respectively, by defining the methods `__add__()`, `__radd__()`, `__iadd__()`, `__mul__()`, `__rmul__()` and `__imul__()`. Sequence types also implement the `__reversed__` method that implements the `reversed()` method that is used for reverse iteration over a sequence. User defined classes can implement these special methods to get the required functionality.

## Emulating Callable Types

Callable types support the function call syntax, `(args)`. Classes that implement the `__call__(self[, args...])` method are callable. User defined classes for which this functionality makes sense can implement this method to make class instances callable. The following example shows a class implementing the `__call__(self[, args...])` method and how instances of this class can be called using the function call syntax.

```
class Account(object):
    num_accounts = 0

    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
        Account.num_accounts += 1

    def __call__(self, arg):
        return "I was called with \'{}\'".format(arg)

    def del_account(self):
        Account.num_accounts -= 1

    def deposit(self, amt):
        self.balance = self.balance + amt

    def withdraw(self, amt):
        self.balance = self.balance - amt

    def inquiry(self):
        return self.balance
```

```
>>> acct = Account()
>>> acct("Testing function call on instance object")
I was called with 'Testing function call on instance object'
```

## Special Methods for comparing objects

User-defined classes can provide custom implementation for the special methods invoked by the five object comparison operators in python, `<`, `>`, `>=`, `<=`, `=` in order to control how these operators work. These special methods are given in the following table.

| Special Method | Description |
|---|---|
| `a.__lt__(self, b)` | a < b |
| `a.__le__(self, b)` | a <= b |
| `a.__eq__(self, b)` | a == b |
| `a.__ne__(self, b)` | a != b |
| `a.__gt__(self, b)` | a > b |
| `a.__ge__(self, b)` | a >= b |

In Python, `x==y` is `True` does not imply that `x!=y` is `False` so `__eq__()` should be defined along with `__ne__()` so that the operators are well behaved. `__lt__()` and `__gt__()`, and `__le__()` and `__ge__()` are each other's reflection while `__eq__()` and `__ne__()` are their own reflection; this means that if a call to the implementation of any of these methods on the left argument returns `NotImplemented`, the reflected operator is is used.

## Special Methods and Attributes for Miscellaneous Customizations

1. `__slots__`: This is a special attribute rather than a method. It is an optimization trick that is used by the interpreter to efficiently store object attributes. Objects by default store all attributes in a dictionary (the `__dict__` attribute) and this is very inefficient when objects with few attributes are created in large numbers. `__slots__` make use of a static iterable that reserves just enough space for each attribute rather than the dynamic `__dict__` attribute. The iterable representing the `__slot__` variable can also be a string made up of the attribute names. The following example shows how `__slots__` works.

```
class Account:
    """base class for representing user accounts"""

    # we can also use __slots__ = "name balance"
    __slots__ = ['name', 'balance']
    num_accounts = 0

    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
        Account.num_accounts += 1

    def del_account(self):
        Account.num_accounts -= 1

    def __getattr__(self, name):
        """handle attribute reference for non-existent attribute"""
        return "Hey I dont see any attribute called {}".format(name)

    def deposit(self, amt):
        self.balance = self.balance + amt

    def withdraw(self, amt):
        self.balance = self.balance - amt

    def inquiry(self):
        return "Name={}, balance={}".format(self.name, self.balance)

>>>acct = Account("obi", 10)
>>>acct.__dict__ # __dict__ attribute is gone
Hey I dont see any attribute called __dict__
>>>acct.x = 10
Traceback (most recent call last):
File "acct.py", line 32, in <module>
  acct.x = 10
AttributeError: 'Account' object has no attribute 'x'
>>>acct.__slots__
['name', 'balance']
```

A few things that are worth noting about __slots__ include the following:

1. If a superclass has the __dict__ attribute then using __slots__ in sub-classes is of no use as the dictionary is available.
2. If __slots__ are used then attempting to assign to a variable not in the __slots__ variable will result in an AttributeError as shown in the previous example.
3. Sub-classes will have a __dict__ even if they inherit from a base class with a __slots__-declaration; subclasses have to define their own __slots__ attribute which must contain only the additional names in order to avoid having the __dict__ for storing names.
4. Subclasses with "variable-length" built-in types as base class cannot have a non-empty __-slots__ variable.
5. __bool__: This method implements the truth value testing for a given class; it is invoked by the built-in operation bool() and should return a True or False value. In the absence of an implementation, __len__() is called and if __len__ is implemented, the object's truth value is considered to be True if result of the call to __len__ is non-zero. If neither __len__() nor __bool__() are defined by a class then all its instances are considered to be True.
6. __repr__ and __str__: These are two closely related methods as they both return string representations for a given object and only differ subtly in the intent behind their creation. Both are invoked by a call to repr and str methods respectively. The __repr__ method implementation should return an unambiguous string representation of the object it is being called on. **Ideally**, the representation that is returned should be an expression that when evaluated by the eval method returns the given object; when this is not possible the representation returned should be as unambiguous as possible. On the other hand, __str__ exists to provide a human readable version of an object; a version that would make sense to some one reading the output but that doesn't necessarily understand the semantics of the language. A very good illustration of how both methods differ is shown below by calling both methods on a data object.

```
>>> import datetime
>>> today = datetime.datetime.now()
>>> str(today)
'2015-07-05 20:55:58.642018' # human readable version of datetime object
>>> repr(today)
'datetime.datetime(2015, 7, 5, 20, 55, 58, 642018)' # eval will return the datetime object
```

   When using string interpolation, %r makes a call to repr while %s makes a call to str.
7. __bytes__: This is invoked by a call to the bytes() built-in and it should return a byte string representation for an object. The byte string should be a bytes object.
8. __hash__: This is invoked by the hash() built-in. It is also used by operations that work on types such as set, frozenset, and dict that make use of object hash values. Providing __-hash__ implementation for user defined classes is an involved and delicate act that should be carried out with care as will be seen. Immutable built-in types are hashable while mutable types such as lists are not. For example, the hash of a number is the value of the number as shown in the following snippet.

```
>>> hash(1)
1
>>> hash(12345)
12345
>>>
```

User defined classes have a default hash value that is derived from their `id()` value. Any `__hash__()` implementation must return an integer and objects that are equal by comparison must have the same hash value so for two object, `a` and `b`, `(a==b and hash(a)==hash(b))` must be true. A few rules for implementing a `__hash__()` method include the following: 1. A class should only define the `__hash__()` method if it also defines the `__eq__()` method.

1. The absence of an implementation for the `__hash__()` method in a class renders its instances unhashable.
2. The interpreter provides user-defined classes with default implementations for `__eq__()` and `__hash__()`. By default, all objects compare unequal except with themselves and `x.__hash__()` returns a value such that (`x == y` and `x is y` and `hash(x) == hash(y)`) is always true. In *CPython*, the default `__hash__()` implementation returns a value derived from the `id()` of the object.
3. Overriding the `__eq__()` method without defining the `__hash__()` method sets the `__hash__()` method to `None` in the class. When the `__hash__()` method of a class is `None`, an instance of the class will raise an appropriate `TypeError` when an attempt is made to retrieve its hash value. The object will also be correctly identified as unhashable when checking `isinstance(obj, collections.Hashable)`.
4. If a class overrides the `__eq__()` and needs to keep the implementation of `__hash__()` from a base class, this must be done explicitly by setting `__hash__ = BaseClass.__hash__`.
5. A class that does not override the `__eq__()` can suppress hash support by setting `__hash__` to `None`. If a class defines its own `__hash__()` method that explicitly raises a `TypeError`, instances of such class will be incorrectly identified as hashable by an `isinstance(obj, collections.Hashable)` test.

## 2.3 A `Vector` class

In this section, a complete example of the use of special methods to emulate built-in types is provided by a `Vector` class. The `Vector` class provides support for performing vector arithmetic operations.

```
# Copyright 2013 Philip N. Klein
class Vec:
    """
    A vector has two fields:
    D - the domain (a set)
    f - a dictionary mapping (some) domain elements to field elements
        elements of D not appearing in f are implicitly mapped to zero
    """
    def __init__(self, labels, function):
        assert isinstance(labels, set)
        assert isinstance(function, dict)
        self.D = labels
        self.f = function


    def __getitem__(self, key):
        """
        Return the value of entry k in v.
        Be sure getitem(v,k) returns 0 if k is not represented in v.f.

        >>> v = Vec({'a','b','c', 'd'},{'a':2,'c':1,'d':3})
        >>> v['d']
        3
        >>> v['b']
        0
        """
        assert key in self.D
        if key in self.f:
            return self.f[key]
        return 0

    def __setitem__(self, key, val):
        """
        Set the element of v with label d to be val.
        setitem(v,d,val) should set the value for key d even if d
        is not previously represented in v.f, and even if val is 0.

        >>> v = Vec({'a', 'b', 'c'}, {'b':0})
        >>> v['b'] = 5
        >>> v['b']
        5
        >>> v['a'] = 1
        >>> v['a']
        1
        >>> v['a'] = 0
        >>> v['a']
        0
        """
        assert key in self.D
        self.f[key] = val
```

```
    def __neg__(self):
        """
        Returns the negation of a vector.

        >>> u = Vec({1,3,5,7},{1:1,3:2,5:3,7:4})
        >>> -u
        Vec({1, 3, 5, 7},{1: -1, 3: -2, 5: -3, 7: -4})
        >>> u == Vec({1,3,5,7},{1:1,3:2,5:3,7:4})
        True
        >>> -Vec({'a','b','c'}, {'a':1}) == Vec({'a','b','c'}, {'a':-1})
        True
        """
        return Vec(self.D, {key:-self[key] for key in self.D})

    def __rmul__(self, alpha):
        """
        Returns the scalar-vector product alpha times v.

        >>> zero = Vec({'x','y','z','w'}, {})
        >>> u = Vec({'x','y','z','w'},{'x':1,'y':2,'z':3,'w':4})
        >>> 0*u == zero
        True
        >>> 1*u == u
        True
        >>> 0.5*u == Vec({'x','y','z','w'},{'x':0.5,'y':1,'z':1.5,'w':2})
        True
        >>> u == Vec({'x','y','z','w'},{'x':1,'y':2,'z':3,'w':4})
        True
        """
        return Vec(self.D, {key : alpha*self[key] for key in self.D })


    def __mul__(self,other):
        #If other is a vector, returns the dot product of self and other
        if isinstance(other, Vec):
            return dot(self,other)
        else:
            return NotImplemented  #  Will cause other.__rmul__(self) to be invoked

    def __truediv__(self,other):  # Scalar division
        return (1/other)*self

    def __add__(self, other):
        """
        Returns the sum of the two vectors.

        Make sure to add together values for all keys from u.f and v.f even if some keys in \
u.f do not
        exist in v.f (or vice versa)

        >>> a = Vec({'a','e','i','o','u'}, {'a':0,'e':1,'i':2})
```

```
        >>> b = Vec({'a','e','i','o','u'}, {'o':4,'u':7})
        >>> c = Vec({'a','e','i','o','u'}, {'a':0,'e':1,'i':2,'o':4,'u':7})
        >>> a + b == c
        True
        >>> a == Vec({'a','e','i','o','u'}, {'a':0,'e':1,'i':2})
        True
        >>> b == Vec({'a','e','i','o','u'}, {'o':4,'u':7})
        True
        >>> d = Vec({'x','y','z'}, {'x':2,'y':1})
        >>> e = Vec({'x','y','z'}, {'z':4,'y':-1})
        >>> f = Vec({'x','y','z'}, {'x':2,'y':0,'z':4})
        >>> d + e == f
        True
        >>> d == Vec({'x','y','z'}, {'x':2,'y':1})
        True
        >>> e == Vec({'x','y','z'}, {'z':4,'y':-1})
        True
        >>> b + Vec({'a','e','i','o','u'}, {}) == b
        True
        """
        assert self.D == other.D
        return Vec(self.D, {key: self[key] + other[key] for key in self.D})

    def __radd__(self, other):
        "Hack to allow sum(...) to work with vectors"
        if other == 0:
            return self

    def __sub__(a,b):
        "Returns a vector which is the difference of a and b."
        return a+(-b)


    def __eq__(self, other):
        """
        Return true iff u is equal to v.

        Consider using brackets notation u[...] and v[...] in your procedure
        to access entries of the input vectors.  This avoids some sparsity bugs.

        >>> Vec({'a', 'b', 'c'}, {'a':0}) == Vec({'a', 'b', 'c'}, {'b':0})
        True
        >>> Vec({'a', 'b', 'c'}, {'a': 0}) == Vec({'a', 'b', 'c'}, {})
        True
        >>> Vec({'a', 'b', 'c'}, {}) == Vec({'a', 'b', 'c'}, {'a': 0})
        True

        Be sure that equal(u, v) checks equalities for all keys from u.f and v.f even if
        some keys in u.f do not exist in v.f (or vice versa)

        >>> Vec({'x','y','z'},{'y':1,'x':2}) == Vec({'x','y','z'},{'y':1,'z':0})
```

```
            False
            >>> Vec({'a','b','c'}, {'a':0,'c':1}) == Vec({'a','b','c'}, {'a':0,'c':1,'b':4})
            False
            >>> Vec({'a','b','c'}, {'a':0,'c':1,'b':4}) == Vec({'a','b','c'}, {'a':0,'c':1})
            False

            The keys matter:
            >>> Vec({'a','b'},{'a':1}) == Vec({'a','b'},{'b':1})
            False

            The values matter:
            >>> Vec({'a','b'},{'a':1}) == Vec({'a','b'},{'a':2})
            False
            """
            assert self.D == other.D
            return all([self[key] == other[key] for key in self.D])

        def is_almost_zero(self):
            s = 0
            for x in self.f.values():
                if isinstance(x, int) or isinstance(x, float):
                    s += x*x
                elif isinstance(x, complex):
                    y = abs(x)
                    s += y*y
                else: return False
            return s < 1e-20

        def __str__(v):
            "pretty-printing"
            D_list = sorted(v.D, key=repr)
            numdec = 3
            wd = dict([(k,(1+max(len(str(k)), len('{0:.{1}G}'.format(v[k], numdec))))) if isinst\
ance(v[k], int) or isinstance(v[k], float) else (k,(1+max(len(str(k)), len(str(v[k]))))) for k in D_\
list])
            s1 = ''.join(['{0:>{1}}'.format(str(k),wd[k]) for k in D_list])
            s2 = ''.join(['{0:>{1}.{2}G}'.format(v[k],wd[k],numdec) if isinstance(v[k], int) or \
isinstance(v[k], float) else '{0:>{1}}'.format(v[k], wd[k]) for k in D_list])
            return "\n" + s1 + "\n" + '-'*sum(wd.values()) +"\n" + s2

        def __hash__(self):
            "Here we pretend Vecs are immutable so we can form sets of them"
            h = hash(frozenset(self.D))
            for k,v in sorted(self.f.items(), key = lambda x:repr(x[0])):
                if v != 0:
                    h = hash((h, hash(v)))
            return h

        def __repr__(self):
            return "Vec(" + str(self.D) + "," + str(self.f) + ")"
```

```
        def copy(self):
            "Don't make a new copy of the domain D"
            return Vec(self.D, self.f.copy())

        def __iter__(self):
            raise TypeError('%r object is not iterable' % self.__class__.__name__)

    if __name__ == "__main__":
        import doctest
        doctest.testmod()
```

## 2.4 Inheritance

Inheritance is one of the basic tenets of object oriented programming and python supports multiple inheritance just like C++. Inheritance provides a mechanism for creating new classes that specialise or modify a base class thereby introducing new functionality. We call the base class the parent class or the super class. An example of a class inheriting from a base class in python is given in the following example.

```
class Account:
    """base class for representing user accounts"""
    num_accounts = 0

    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
        Account.num_accounts += 1

    def del_account(self):
        Account.num_accounts -= 1

    def __getattr__(self, name):
        """handle attribute reference for non-existent attribute"""
        return "Hey I dont see any attribute called {}".format(name)

    def deposit(self, amt):
        self.balance = self.balance + amt

    def withdraw(self, amt):
        self.balance = self.balance - amt

    def inquiry(self):
        return "Name={}, balance={}".format(self.name, self.balance)

class SavingsAccount(Account):

    def __init__(self, name, balance, rate):
        super().__init__(name, balance)
```

```
        self.rate = rate


    def __repr__(self):
            return "SavingsAccount({}, {}, {})".format(self.name, self.balance, self.rate)


>>>acct = SavingsAccount("Obi", 10, 1)
>>>repr(acct)
SavingsAccount(Obi, 10, 1)
```

## The `super` keyword

The `super` keyword plays an integral part in python inheritance. In a single inheritance hierarchy, the `super` keyword is used to refer to the parent/super class without explicitly naming it. This is similar to the `super` method in Java. This comes into play when overriding a method and there is a need to also call the parent version of such method as shown in the above example in which the `__init__` method in the `SavingsAccount` class is overridden but the `__init__` method of the parent class is also called using the `super` method. The `super` keyword plays a more integral role in python inheritance when a multiple inheritance hierarchy exists.

## Multiple Inheritance

In multiple inheritance, a class can have multiple parent classes. This type of hierarchy is strongly discouraged. One of the issues with this kind of inheritance is the complexity involved in properly resolving methods when called. Imagine a class, `D`, that inherits from two classes, `B` and `C` and there is a need to call a method from the parent classes however both parent classes implement the same method. *How is the order in which classes are searched for the method determined ?* A *Method Resolution Order* algorithm determines how a method is found in a class or any of the class' base classes. In Python, the resolution order is calculated at class definition time and stored in the class `__dict__` as the `__mro__` attribute. To illustrate this, imagine a class hierarchy with multiple inheritance such as that showed in the following example.

```
>>> class A:
...     def meth(self): return "A"
...
>>> class B(A):
...     def meth(self): return "B"
...
>>> class C(A):
...     def meth(self): return "C"
...
>>> class D(B, C):
...     def meth(self): return "X"
...
```

```
    >>>
    >>> D.__mro__
    (<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class \
'object'>)
    >>>
```

To obtain an `mro`, the interpreter method resolution algorithm carries out a left to right depth first listing of all classes in the hierarchy. In the trivial example above, this results in the following class list, `[D, B, A, C, A, object]`. Note that all objects will inherit from the root `object` class if no parent class is supplied during class definition. Finally, for each class that occurs multiple times, all occurrences are removed except the last occurrence resulting in an *mro* of `[D, B, C, A, object]` for the previous class hierarchy. This result is the order in which classes would be searched for attributes for a given instance of `D`.

## Cooperative method calls with `super`

This section will show the power of the `super` keyword in a multiple inheritance hierarchy. The class hierarchy from the previous section is used. This example is from the excellent write up[1] by Guido Van Rossum on *Type Unification*. Imagine that `A` defines a method that is overridden by `B`, `C` and `D`. Suppose that there is a requirement that all the methods are called; the method may be a `save` method that saves an attribute for each type it is defined for, so missing any call will result in some unsaved data in the hierarchy. A combination of `super` and `__mro__` provide the ammunition for solving this problem. This solution is referred to as the *call-next* method by Guido van Rossum and is shown in the following snippet:

```
class A(object):
    def meth(self):
        "save A's data"
        print("saving A's data")

class B(A):
    def meth(self):
        "save B's data"
        super(B, self).meth()
        print("saving B's data")

class C(A):
    def meth(self):
        "save C's data"
        super(C, self).meth()
        print("saving C's data")

class D(B, C):
    def meth(self):
        "save D's data"
```

[1] https://www.python.org/download/releases/2.2.3/descrintro/

```
        super(D, self).meth()
        print("saving D's data")
```

When *self.meth()* is called by an instance of `D` for example, `super(D, self).meth()` will find and call `B.meth(self)`, since B is the first base class following `D` that defines `meth` in `D.__mro__`. Now in `B.meth`, `super(B, self).m()` is called and since `self` is an instance of `D`, the next class after `B` is `C` (`__mro__` is `[D, B, C, A]`) and the search for a definition of `meth` continues here. This finds `C.meth` which is called, and which in turn calls `super(C, self).m()`. Using the same *MRO*, the next class after `C` is `A`, and thus `A.meth` is called. This is the original definition of m, so no further *super()* call is made at this point. Using `super` and method resolution order, the interpreter has been able to find and call all version of the `meth` method implemented by each of the classes in the hierarchy. However, multiple inheritance is best avoided because for more complex class hierarchies, the calls may be way more complicated than this.

# 2.5 Static and Class Methods

All methods defined in a class by default operate on instances. However, one can define static or class methods by decorating such methods with the corresponding `@staticmethod` or `@classmethod` decorators.

## Static Methods

Static methods are normal functions that exist in the name-space of a class. Referencing a static method from a class shows that rather than an *unbound* method type, a *function* type is returned as shown below:

```
class Account(object):
    num_accounts = 0

    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
        Account.num_accounts += 1

    def del_account(self):
        Account.num_accounts -= 1

    def deposit(self, amt):
        self.balance = self.balance + amt

    def withdraw(self, amt):
        self.balance = self.balance - amt

    def inquiry(self):
```

```
        return "Name={}, balance={}".format(self.name, self.balance)

    @staticmethod
    def static_test_method():
        return "Current Account"

>>> Account.static_test_method
<function Account.static_test_method at 0x101b846a8>
```

To define a static method, the `@staticmethod` decorator is used and such methods do not require the `self` argument. Static methods provide a mechanism for better organization as code related to a class are placed in that class and can be overridden in a sub-class as needed. Unlike ordinary class methods that are wrappers around the actual underlying functions, static methods return the underlying functions without any modification when used.

## Class Methods

Class methods as the name implies operate on classes themselves rather than instances. Class methods are created using the `@classmethod` decorator with the `class` rather than `instance` passed as the first argument to the method.

```
import json

class Account(object):
    num_accounts = 0

    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
        Account.num_accounts += 1

    def del_account(self):
        Account.num_accounts -= 1

    def deposit(self, amt):
        self.balance = self.balance + amt

    def withdraw(self, amt):
        self.balance = self.balance - amt

    def inquiry(self):
        return "Name={}, balance={}".format(self.name, self.balance)

    @classmethod
    def from_json(cls, params_json):
            params = json.loads(params_json)
        return cls(params.get("name"), params.get("balance"))
```

```
    @staticmethod
    def type():
        return "Current Account"
```

A motivating example of the usage of class methods is as a *factory* for object creation. Imagine data for the `Account` class comes in different formats such as *tuples, json string* etc. It is not possible to define multiple `__init__` methods in a class so class methods come in handy for such situations. In the `Account` class defined above for example, there is a requirement to initialize an account from a *json* string object so we define a class factory method, `from_json` that takes in a json string object and handles the extraction of parameters and creation of the account object using the extracted parameters. Another example of a class method in action as a factory method is the `dict.fromkeys`[2] methods that is used for creating *dict* objects from a sequence of supplied keys and value.

## 2.6 Descriptors and Properties

Descriptors are an esoteric but integral part of the python programming language. They are used widely in the core of the python language and a good grasp of descriptors provides a python programmer with a deeper understanding of the language. To set the stage for the discussion of descriptors, some scenarios that a programmer may encounter are described; this is followed by an explanation of descriptors and how they provide elegant solutions to these scenarios.

1. Consider a program in which some rudimentary type checking of object data attributes needs to be enforced. Python is a dynamic languages so does not support type checking but this does not prevent anyone from implementing a version of type checking regardless of how rudimentary it may be. The conventional way to go about type checking object attributes may take the following form.

    ```
    def __init__(self, name, age):
        if isinstance(name, str):
            self.name = name
        else:
            raise TypeError("Must be a string")
        if isinstance(age, int):
            self.age = age
        else:
            raise TypeError("Must be an int")
    ```

    The above method maybe feasible for enforcing such type checking for one or two data attributes but as the attributes increase in number it gets cumbersome. Alternatively, a `type_check(type, val)` function could be defined and this will be called in the `__init__` method before assignment; but this cannot be elegantly applied when the attribute value is set after initialization. A quick solution that comes to mind is the `getters` and `setters` present in Java but that is un-pythonic and cumbersome.

[2] https://docs.python.org/3/library/stdtypes.html#dict.fromkeys

2. Consider a program that needs object attributes to be read-only once initialized. One could also think of ways of implementing this using Python special methods but once again such implementation could be unwieldy and cumbersome.
3. Finally, consider a program in which the attribute access needs to be customized. This maybe to log such attribute access or to even perform some kind of transformation of the attribute for example. Once again, it is not too difficult to come up with a solution to this although such solution maybe unwieldy and not reusable.

All the above mentioned issues are all linked together by the fact that they are all related to attribute references. Attribute access is trying to be customized.

## Enter Python Descriptors

Descriptors provide elegant, simple, robust and re-usable solutions to the above listed issues. Simply put, a *descriptor* is an **object** that represents the value of an attribute. This means that if an account object has an attribute `name`, a descriptor is another object that can be used to represent the value held by that attribute, `name`. Such an object implements the `__get__`, `__set__` or `__delete__` special methods of the descriptor protocol. The signature for each of these methods is shown below:

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

Objects implementing only the `__get__` method are non-data descriptors so they can only be read from after initialization while objects implementing the `__get__` and `__set__` are data descriptors meaning that such descriptor objects are writable.

To get a better understanding of descriptors descriptor based solutions are provided to the issues mentioned in the previous section. Implementing type checking on an object attribute using descriptors is a simple and straightforward task. A decorator implementing this type checking is shown in the following snippet.

```
class TypedAttribute:

    def __init__(self, name, type, default=None):
        self.name = "_" + name
        self.type = type
        self.default = default if default else type()

    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default)

    def __set__(self,instance,value):
        if not isinstance(value,self.type):
```

```
            raise TypeError("Must be a %s" % self.type)
        setattr(instance,self.name,value)

    def __delete__(self,instance):
        raise AttributeError("Can't delete attribute")


class Account:
    name = TypedAttribute("name",str)
    balance = TypedAttribute("balance",int, 42)

>> acct = Account()
>> acct.name = "obi"
>> acct.balance = 1234
>> acct.balance
1234
>> acct.name
obi
# trying to assign a string to number fails
>> acct.balance = '1234'
TypeError: Must be a <type 'int'>
```

In the example, a descriptor, `TypedAttribute` is implemented and this descriptor class enforces rudimentary type checking for any attribute of a class which it is used to represent. It is important to note that descriptors are effective in this kind of case only when defined at the class level rather than instance level i.e. in `__init__` method as shown in the example above.

Descriptors are integral to the Python language. Descriptors provide the mechanism behind properties, static methods, class methods, `super` and a host of other functionality in Python classes. In fact, descriptors are the first type of object searched for during an attribute reference. When an object is referenced, a reference, `b.x`, is transformed into `type(b).__dict__['x'].__get__(b, type(b))`. The algorithm then searches for the attribute in the following order.

1. `type(b).__dict__` is searched for the attribute name and if a data descriptor is found, the result of calling the descriptor's `__get__` method is returned. If it is not found, then all base classes of `type(b)` are searched in the same way.
2. `b.__dict__` is searched and if attribute name is found here, it is returned.
3. `type(b).__dict` is searched for a non-data descriptor with given attribute name and if found it is returned,
4. If the name is not found, an `AttributeError` is raised or `__getattr__()` is called if provided.

This precedence chain can be overridden by defining custom `__getattribute__` methods for a given object class (the precedence defined above is contained in the default `__getattribute__` provided by the interpreter).

With a firm understanding of the mechanics of descriptors, it is easy to implement elegant solutions to the second and third issues raised in the previous section. Implementing a read only attribute

with descriptors becomes a simple case of implementing a non-data descriptor *i.e descriptor with no* `__set__` *method*. To solve the custom access issue, whatever functionality is required is added to the `__get__`and `__set__` methods respectively.

## Class Properties

Defining descriptor classes each time a descriptor is required is cumbersome. Python ***properties*** provide a concise way of adding data descriptors to attributes. A property signature is given below:

```
property(fget=None, fset=None, fdel=None, doc=None) -> property attribute
```

`fget`, `fset` and `fdel` are the *getter, setter and deleter* methods for such class attributes. The process of creating properties is illustrated with the following example.

```
class Accout(object):
    def __init__(self):
        self._acct_num = None

    def get_acct_num(self):
        return self._acct_num

    def set_acct_num(self, value):
        self._acct_num = value

    def del_acct_num(self):
        del self._acct_num

    acct_num = property(get_acct_num, set_acct_num, del_acct_num, "Account number property.")
```

If `acct` is an instance of `Account`, `acct.acct_num` will invoke the getter, `acct.acct_num = value` will invoke the setter and `del acct_num.acct_num` will invoke the deleter.

The property object and functionality can be implemented in python as illustrated in Descriptor How-To Guide[3] using the descriptor protocol as shown below :

[3] https://docs.python.org/2/howto/descriptor.html

```
class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)

    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel, self.__doc__)

    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel, self.__doc__)

    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel, self.__doc__)
```

Python also provides the `@property` decorator that can be used to create read only attributes. A property object has getter, setter, and deleter decorator methods that can be used to create a copy of the property with the corresponding *accessor* function set to the decorated function. This is best explained with an example:

```
class C(object):
    def __init__(self):
        self._x = None

    @property
     # the x property. the decorator creates a read-only property
    def x(self):
        return self._x

    @x.setter
    # the x property setter makes the property writeable
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

If a property is read-only then the `setter` method is left out.

An understanding of descriptors puts us in a better corner to understand what actually goes on during a method call. Note that methods are stored as ordinary functions in a class dictionary as shown in the following snippet.

```
>>>Account.inquiry
<function Account.inquiry at 0x101a3e598>
>>>
```

However, object methods are of bound method type as shown in the following snippet.

```
>>> x = Account("nkem", 10)
>>> x.inquiry
<bound method Account.inquiry of <Account object at 0x101a3c588>>
```

To understand how this transformation takes place, note that a bound method is just a thin wrapper around the class function. Functions are descriptors because they have the `__get__` method attribute so a reference to a function will result in a call to the `__get__` method of the function and this returns the desired type, the function itself or a bound method, depending on whether this reference is from a class or from an instance of the class. It is not difficult to imagine how static and class methods maybe implemented by the function descriptor and this is left to the reader to come up with.

## 2.7 Abstract Base Classes

Sometimes, it is necessary to enforce a contract between classes in a program. For example, it may be necessary for all classes to implement a set of methods. This is accomplished using interfaces and

abstract classes in statically typed languages like Java. In Python, a base class with default methods may be implemented and then all other classes within the set inherit from the base class. However, there is a requirement for each each subclass to have its own implementation and this rule needs to be enforced. All the needed methods can be defined in a base class with each of them having an implementation that raises the `NotImplementedError` exception. All subclasses then have to override these methods in order to use them. However this does not still solve the problem fully. It is possible that some subclasses may not implement some of these method and it would not be known till a method call was attempted at runtime.

Consider another situation of a proxy object that passes method calls to another object. Such a proxy object may implement all required methods of a type via its proxied object, but an `isinstance` test on such a proxy object for the proxied object will fail to produce the correct result.

Python's *Abstract base classes* provide a simple and elegant solution to these issues mentioned above. The abstract base class functionality is provided by the *abc* module. This module defines a meta-class (we discuss meta-classes in the chapter on meta-programming) and a set of decorators that are used in the creation of abstract base classes. When defining an abstract base class we use the `ABCMeta` meta-class from the `abc` module as the meta-class for the abstract base class and then make use of the `@abstractmethod` and `@abstractproperty` decorators to create methods and properties that must be implemented by non-abstract subclasses. If a subclass doesn't implement any of the abstract methods or properties then it is also an abstract class and cannot be instantiated as illustrated below:

```
from abc import ABCMeta, abstractmethod

class Vehicle(object):
    __meta-class__ = ABCMeta

    @abstractmethod
    def change_gear(self):
        pass

    @abstractmethod
    def start_engine(self):
        pass


class Car(Vehicle):

    def __init__(self, make, model, color):
        self.make = make
        self.model = model
        self.color = color

# abstract methods not implemented
>>> car = Car("Toyota", "Avensis", "silver")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Car with abstract methods change_gear, start_engine
>>>
```

Once, a class implements all abstract methods then that class becomes a concrete class and can be instantiated by a user.

```
from abc import ABCMeta, abstractmethod

class Vehicle(object):
    __meta-class__ = ABCMeta

    @abstractmethod
    def change_gear(self):
        pass

    @abstractmethod
    def start_engine(self):
        pass


class Car(Vehicle):

    def __init__(self, make, model, color):
        self.make = make
        self.model = model
        self.color = color

    def change_gear(self):
        print("Changing gear")

    def start_engine(self):
        print("Changing engine")

>>> car = Car("Toyota", "Avensis", "silver")
>>> print(isinstance(car, Vehicle))
True
```

Abstract base classes also allow existing classes to register as part of its hierarchy but it performs no check on whether such classes implement all the methods and properties that have been marked as abstract. This provides a simple solution to the second issue raised in the opening paragraph. Now, a proxy class can be registered with an abstract base class and `isinstance` check will return the correct answer when used.

```
from abc import ABCMeta, abstractmethod

class Vehicle(object):
    __meta-class__ = ABCMeta

    @abstractmethod
    def change_gear(self):
        pass

    @abstractmethod
    def start_engine(self):
        pass


class Car(object):

    def __init__(self, make, model, color):
        self.make = make
        self.model = model
        self.color = color

>>> Vehicle.register(Car)
>>> car = Car("Toyota", "Avensis", "silver")
>>> print(isinstance(car, Vehicle))
True
```

Abstract base classes are used a lot in python library. They provide a mean to group python objects such as number types that have a relatively flat hierarchy. The `collections` module also contains abstract base classes for various kinds of operations involving sets, sequences and dictionaries. Whenever we want to enforce contracts between classes in python just as interfaces do in Java, abstract base classes is the way to go.