Advanced Python

– Bay Area Python3.x Rigorous Indulgence Group (BayPRIGgies)

Veera V Pendyala
veera.pendyala@gmail.com

October 23, 2016

Introduction

Introduction

Basic understanding of Python Programing is a prerequisite.

All the examples are worked out using Python 3.5.

What do you learn?

- Understand the underlying concepts of the Python OOP paradigm.
- Introduction to Python 3.x

Built-in Functions

__map__

```
>>> def square(number):
...     return number * number

>>> list(map(square, [1, 2, 3, 4, 5, 6]))
[1, 4, 9, 16, 25, 36]

>>> # Can also be written as a list comprehension.
>>> [square(x) for x in [1, 2, 3, 4, 5, 6]]
[1, 4, 9, 16, 25, 36]

>>> # Another Example for __map__
>>> list(map(pow,[2, 3, 4], [10, 11, 12]))
[1024, 177147, 16777216]
```

__filter__

```
>>> def square(number):
...     return number * number

>>> list(map(square, [0, 1, 2, 3, 4, 5, 6, 0]))
[0, 1, 4, 9, 16, 25, 36, 0]

>>> list(filter(square, [0, 1, 2, 3, 4, 5, 6, 0]))
[1, 4, 9, 16, 25, 36]

>>> # Can also be written as a list comprehension.
>>> [square(x) for x in [0, 1, 2, 3, 4, 5, 6, 0] if x]
[1, 4, 9, 16, 25, 36]
```

all

```
>>> all([True, False])
False
>>> all([True, True])
True

>>> def is_positive(number):
...     if number >= 0:
...         return True
...     return False

>>> all([is_positive(x) for x in [1, 2]])
True
>>> all([is_positive(x) for x in [1, -1]])
False
>>> all([is_positive(x) for x in [-1, -2]])
False
```

any

```
>>> any([True, False])
True
>>> any([False, False])
False

>>> def is_positive(number):
...     if number >= 0:
...         return True
...     return False

>>> any([is_positive(x) for x in [-1, 1]])
True
>>> any([is_positive(x) for x in [-1, -2]])
False
```

getattr

```
>>> class Foo(object):
...     def __init__(self):
...         self.value_1 = 5
...     def test_method(self):
...         return 10
...
>>> foo = Foo()
>>> getattr(foo, 'test_method')
<bound method Foo.test_method of <__main__.Foo object at 0x101f1f208>>
>>> getattr(foo, 'test_method')()
10
>>> getattr(foo, 'value_1')
5
```

hasattr

Example

```
>>> class Foo(object):
...     def __init__(self):
...         self.value_1 = 5
...     def test_method(self):
...         return 10
...
>>> foo = Foo()
>>> hasattr(foo, 'test_method')
True
>>> hasattr(foo, 'test_method_not_there')
False
>>> hasattr(foo, 'value_1')
True
```

setattr

Example

```
>>> class Foo(object):
...     def __init__(self):
...         self.value_1 = 5
...     def test_method(self):
...         return 10
...
>>> foo = Foo()
>>> setattr(foo, 'value_1', 15)
>>> getattr(foo, 'value_1')
15
>>> foo.value_1
15
>>> setattr(foo, 'value_2', 25)
>>> foo.value_2
25
```

setattr

Example

```
>>> def test_method_2(var):
...     return var
>>> setattr(foo, 'test_method_2', test_method_2)
>>> getattr(foo, 'test_method_2')(20)
20
>>> foo.test_method_2(40)
40
```

callable

Example

```
>>> class Foo(object):
...     def __init__(self):
...         self.not_callable = None
...     def foo_method(self):
...         pass
...     @property
...     def test_property(self):
...         pass
```

callable

On class

```
>>> callable(Foo)
True
>>> callable(Foo.foo_method)
True
>>> callable(Foo.test_property)
False
>>> callable(Foo.not_callable)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Foo' has no attribute 'not_callable'
```

callable

On class object

```
>>> foo = Foo()
>>> callable(foo.foo_method)
True
>>> callable(foo.not_callable)
False
>>> callable(foo.test_property)
False
```

isinstance

Example

```
>>> class Foo(object):
...     def __init__(self):
...         self.not_callable = None
...     def foo_method(self):
...         pass
...     @property
...     def test_property(self):
...         pass
...
>>> foo = Foo()
>>> type(Foo)
<class 'type'>
>>> isinstance(foo, Foo)
True
```

issubclass

Example

```
>>> class Foo(object):
...     pass
...
>>> class Bar(Foo):
...     pass
...
>>> isinstance(Foo, Bar)
False
>>> isinstance(Bar, Foo)
False
>>> issubclass(Foo, Bar)
False
>>> issubclass(Bar, Foo)
True
```

isinstance

Example

```
>>> foo = Foo()
>>> bar = Bar()
>>> isinstance(foo, Foo)
True
>>> isinstance(bar, Foo)
True
>>> isinstance(bar, Bar)
True
>>> isinstance(foo, Bar)
False
```

Closure

Closure

A **CLOSURE** is a function object that remembers values in enclosing scopes regardless of whether those scopes are still present in memory. If you have ever written a function that returned another function, you probably may have used closures even without knowing about them.

There is a saying in computer science that “a class is **data with operations attached** while a closure is **operations with data attached.**” – David

closure

```
>>> def generate_adder_func(n):
...     print("id(n): %x" % id(n))
...     def adder(x):
...         return n+x
...     print("id(adder): %x" % id(adder))
...     return adder

>>> add_to_5 = generate_adder_func(5)
id(n): 1002148f0
id(adder): 1010758c8
>>> repr(add_to_5)
<function generate_adder_func.<locals>.adder at 0x1010758c8>
```

Closure

```
>>> del generate_adder_func
>>> 'Value is: %d' % add_to_5(5)
Value is: 10
```

```
>>> add_to_5.__closure__
(<cell at 0x1019382e8: int object at 0x1002148f0>,)
>>> type(add_to_5.__closure__[0])
<class 'cell'>
>>> add_to_5.__closure__[0].cell_contents
5
```

As you can see, the **__closure__** attribute of the function **add_to_5** has a reference to int object at **0x1002148f0** which is none other than **n** (which was defined in **generate_adder_func**)

Closure

In case you're wondering, every function object has **__closure__** attribute. If there is not data for the closure, the **__closure__** attribute will just be **None**. For example

```
>>> def foo():
...     pass

>>> repr(foo); repr(foo.__closure__)
'<function foo at 0x101f22f28>'
'None'
```

Decorator

@decorator

Same functionality can be achieved without using the decorator syntax

```
def bar(foo):
    pass
@bar
def foo():
    pass
```

Is equivalent to

```
def foo():
    pass
foo = bar(foo)
```

@decorator

```
>>> def my_decorator(func_obj):
...     return func_obj

>>> @my_decorator
... def my_func():
...     print("Hello world!")

>>> my_func()
Hello world!
>>> my_func
<function my_func at 0x101f3e488>
```

@decorator

NOT getting desired output!

```
>>> def my_decorator(func_obj):
...     print('This is decorator')
...     return func_obj
>>> @my_decorator
... def my_func():
...     print('Hello world!')
This is decorator
>>> my_func()
Hello world!
>>> my_func
<function my_func at 0x101f3e2f0>
```

@decorator

Desired output!

```
>>> import time
>>> def my_decorator(func_obj):
...     def wrapped(*args, **kwargs):
...         print('Start time: %d' % time.time())
...         return_obj = func_obj(*args, **kwargs)
...         print('End time: %d' % time.time())
...         return return_obj
...     return wrapped
>>> @my_decorator
... def my_func(name):
...     print('Hello world! %s' % name)
...     return 'Done'
>>> my_func('Subbu')
Start time: 1474754409
Hello world! Subbu
End time: 1474754409
'Done'
```

@decorator

Doc strings of functions!

```
>>> def foo():
...     ''' This is foo '''
...     pass
...
>>> foo.__doc__
' This is foo '
```

@decorator

Let's check them for decorated functions.

```
>>> import time
>>> def my_decorator(func_obj):
...     def wrapped(*args, **kwargs):
...         print('Start time: %d' % time.time())
...         return_obj = func_obj(*args, **kwargs)
...         print('End time: %d' % time.time())
...         return return_obj
...     return wrapped
>>> @my_decorator
... def my_func(name):
...     ''' My Function '''
...     print("Hello world! %s" % name)
...     return 'Done'
>>> my_func.__doc__
>>>
>>> my_func.__name__
'wrapped'
```

@decorator

Important metadata such as the name, doc string, annotations, and calling signature are preserved.

```
>>> import time
>>> from functools import wraps
>>> def my_decorator(func_obj):
...     ''' My Decorator '''
...     @wraps(func_obj)
...     def wrapped(*args, **kwargs):
...         print('Start time: %d' % time.time())
...         return_obj = func_obj(*args, **kwargs)
...         print('End time: %d' % time.time())
...         return return_obj
...     return wrapped
>>> @my_decorator
... def my_func(name):
...     ''' My Function '''
...     print("Hello world! %s" % name)
...     return 'Done'
>>> my_func.__doc__
' My Function '
>>> my_func.__name__
'my_func'
```

@decorator

Class decorator

```
>>> import time
>>> class ClassDecorator(object):
...     def __init__(self, func):
...         self.func = func
...     def __call__(self, *args, **kwargs):
...         print('Start: %s' % time.time())
...         ret_obj = self.func.__call__(self.obj, *args, **kwargs)
...         print('End: %s' % time.time())
...         return ret_obj
...     def __get__(self, instance, owner):
...         self.cls = owner
...         self.obj = instance
...         return self.__call__
>>> class Test(object):
...     def __init__(self):
...         self.name = 'Subbu'
...     @ClassDecorator
...     def test_method(self):
...         ''' Test method '''
...         return self.name
>>> test_obj = Test()
>>> print('Hello %s' % test_obj.test_method())
Start: 1474777873.204947
End: 1474777873.205015
Hello Subbu
```

Memoizing with function Decorator

@decorator

Let us look at the classic factorial function

```
>>> def factorial(n):
...     if n == 1:
...         return 1
...     else:
...         return n * factorial(n-1)

>>> factorial(5)
120
```

TODO: Look for decorators with arguments.

Solution for factorial

```
>>> def memoize(f):
...     memo = {}
...     def helper(x):
...         if x not in memo:
...             print('Finding the value for %d' % x)
...             memo[x] = f(x)
...         print('Factorial for %d is %d' % (x, memo[x]))
...         return memo[x]
...     return helper
...
>>> @memoize
... def factorial(n):
...     if n == 1:
...         return 1
...     else:
...         return n * factorial(n-1)
...
>>> factorial(5)
Finding the value for 5
Finding the value for 4
Finding the value for 3
Finding the value for 2
Finding the value for 1
Factorial for 1 is 1
Factorial for 2 is 2
Factorial for 3 is 6
Factorial for 4 is 24
Factorial for 5 is 120
120
```

Solution for fibonacci

```
>>> def memoize(f):
...     memo = {}
...     def helper(x):
...         if x not in memo:
...             memo[x] = f(x)
...         return memo[x]
...     return helper
...
>>> @memoize
... def fib(n):
...     if n == 0:
...         return 0
...     elif n == 1:
...         return 1
...     else:
...         return fib(n-1) + fib(n-2)
...
>>> fib(10)
55
```

Generators

Simple Generator

Generators are data producers

```
>>> def generator():
...     while True:
...         yield 1
>>> a = generator()
>>> next(a)
1
```

Round robin Generator

Generators are data producers

```
>>> def round_robin():
...     while True:
...         yield from [1, 2, 3, 4]
>>> a = round_robin()
>>> a
<generator object round_robin at 0x101a89678>
>>> next(a)
1
>>> next(a)
2
>>> next(a)
3
>>> next(a)
4
>>> next(a)
1
>>> next(a)
2
>>> next(a)
3
>>> next(a)
4
```

Coroutine

Simple Coroutine

Coroutines are data consumers.

```
>>> def coroutine():
...     while True:
...         val = (yield)
...         print(val)

>>> a = coroutine()
>>> next(a)
>>> a.send(1)
1
>>> a.send(2)
2
>>> a.close()

>>> b = coroutine()
>>> b.send(None)
>>> b.send(1)
1
>>> b.send(2)
2
>>> b.close()
```

Coroutine

All coroutines must be “primed” by first calling **next** or **send(None)**

This advances execution to the location of the first **yield** expression.

For the first **yield** it is ready to receive the values.

Call **close** to shut down the coroutine

Class Properties

@property

Simple getter and setter.

```
>>> class Foo(object):
...     def __init__(self, x):
...         self.__x = x
...     def getX(self):
...         return self.__x
...     def setX(self, x):
...         self.__x = x
...
>>> foo = Foo(5)
>>> foo.getX()
5
>>> foo.setX(10)
>>> foo.getX()
10
```

@property

Python way of defining is much simpler.

```
>>> class Foo(object):
...     def __init__(self,x):
...         self.x = x
...
>>> foo = Foo(5)
>>> foo.x
5
>>> foo.x = 10
>>> foo.x
10
```

Where is data ENCAPSULATION?

getX and **setX** in our starting example did was "**getting the data through**" without doing anything, no checks nothing.

What happens, if we want to change the implementation in the future? Lets change the implementation, as following:

- The attribute **x** can have values between **0** and **1000**
- If a value larger than **1000** is assigned, **x** should be set to **1000**
- **x** should be set to **0**, if the value is less than **0**

@property

```
>>> class Foo(object):
...     def __init__(self,x):
...         self.__x = x
...     def getX(self):
...         return self.__x
...     def setX(self, x):
...         if x < 0:
...             self.__x = 0
...         elif x > 1000:
...             self.__x = 1000
...         else:
...             self.__x = x
...     def delX(self):
...         del self.__x

>>> foo = Foo(5)
>>> foo.getX()
5
>>> foo.setX(1010)
>>> foo.getX()
1000
>>> foo.setX(-1)
>>> foo.getX()
0
```

@property

People recommended to use only private attributes with **getters** and **setters**, so that they can change the implementation without having to change the interface.

Let's assume we have designed our class with the public attribute and no methods. People have already used it a lot and they have written code like this:

```
>>> from foo_module import foo
>>> foo = Foo(100)
>>> foo.x = 1001
```

@property

```
>>> class Foo(object):
...     def __init__(self):
...         self.__x = None
...     def getX(self):
...         return self.__x
...     def setX(self, x):
...         if x < 0:
...             self.__x = 0
...         elif x > 1000:
...             self.__x = 1000
...         else:
...             self.__x = x
...     def delX(self):
...         del self.__x
...     x = property(getX,   setX,   delX,    "I'm the 'x' property.")
...                # (getter, setter, deleter, doc_string)
>>> foo = Foo()
>>> foo
<__main__.Foo object at 0x1011aaef0>
>>> f
>>> foo.x
100
>>> foo.x = 1001
>>> foo.x
1000
>>> foo.x = -1
>>> foo.x
0
```

@property

Deleting the value

```
>>> del foo.x
>>> foo.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in getX
AttributeError: 'Foo' object has no attribute '_Foo__x'
>>> foo
<__main__.Foo object at 0x1011aaef0>
>>> foo.x = 2000
>>> foo.x
1000
```

@property

The **@property** decorator turns the method into a **getter** for a read-only attribute with the same name.

A property object has **getter**, **setter**, and **deleter** methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```
>>> class Foo(object):
...     def __init__(self):
...         self.__x = None
...     @property
...     def x(self):
...         return self.__x
...     @x.setter
...     def x(self, x):
...         if x < 0:
...             self.__x = 0
...         elif x > 1000:
...             self.__x = 1000
...         else:
...             self.__x = x
...     @x.deleter
...     def x(self):
...         del self.__x
```

@property

This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original property (**x** in this case.)

The returned property object also has the attributes **fget**, **fset**, and **fdel** corresponding to the constructor arguments.

```
>>> foo = Foo()
>>> foo
<__main__.Foo object at 0x1011aaf60>
>>> foo.x = 1001
>>> foo.x
1000
>>> foo.x = 10
>>> foo.x
10
>>> foo.x = -1
>>> foo.x
0
>>> del foo.x
>>> foo.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in x
AttributeError: 'Foo' object has no attribute '_Foo__x'
>>> foo
<__main__.Foo object at 0x1011aaf60>
```

Class Attributes

Attributes

Instance attributes are owned by the specific instances of a class.

This means for two different instances the instance attributes are usually different.

Now, let us define attributes at the class level.

Class attributes are attributes which are owned by the class itself.

They will be shared by all the instances of the class.

Attributes

```
>>> class Foo(object):
...     val = 5
>>> a = Foo()
>>> a.val
5
>>> a.val = 10
>>> a.val
10
>>> b = Foo()
>>> b.val
5
>>> Foo.val
5
>>> Foo.val = 100
>>> Foo.val
100
>>> b.val
100
>>> a.val
10
```

We can access a class attribute via instance or via the class name. As you can see in the example above that we need no instance.

Attributes

Python's class attributes and object attributes are stored in separate dictionaries, as we can see here:

```
>>> a.__dict__
{'val': 10}
>>> b.__dict__
{}
>>> import pprint
>>> pprint.pprint(Foo.__dict__)
mappingproxy({'__dict__': <attribute '__dict__' of 'Foo' objects>,
              '__doc__': None,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Foo' objects>,
              'val': 100})
>>> pprint.pprint(a.__class__.__dict__)
mappingproxy({'__dict__': <attribute '__dict__' of 'Foo' objects>,
              '__doc__': None,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Foo' objects>,
              'val': 100})
>>> pprint.pprint(b.__class__.__dict__)
mappingproxy({'__dict__': <attribute '__dict__' of 'Foo' objects>,
              '__doc__': None,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Foo' objects>,
              'val': 100})
```

Attributes

Demonstrate to count instances with class attributes.

```
>>> class Foo(object):
...     instance_count = 0
...     def __init__(self):
...         Foo.instance_count += 1
...     def __del__(self):
...         Foo.instance_count -= 1
...
>>> a = Foo()
>>> print('Instance count %s' % Foo.instance_count)
Instance count 1
>>> b = Foo()
>>> print('Instance count %s' % Foo.instance_count)
Instance count 2
>>> del a
>>> print('Instance count %s' % Foo.instance_count)
Instance count 1
>>> del b
>>> print('Instance count %s' % Foo.instance_count)
Instance count 0
```

Attributes

Demonstrate to count instance with class attributes using **type(self)**

```
>>> class Foo(object):
...     instance_count = 0
...     def __init__(self):
...         type(self).instance_count += 1
...     def __del__(self):
...         type(self).instance_count -= 1
...
>>> a = Foo()
>>> print('Instance count %s' % Foo.instance_count)
Instance count 1
>>> b = Foo()
>>> print('Instance count %s' % Foo.instance_count)
Instance count 2
>>> del a
>>> print('Instance count %s' % Foo.instance_count)
Instance count 1
>>> del b
>>> print('Instance count %s' % Foo.instance_count)
Instance count 0
```

Attributes

Demonstrate to count instance ERROR using private class attributes.

```
>>> class Foo(object):
...     __instance_count = 0
...     def __init__(self):
...         type(self).__instance_count += 1
...     def foo_instances(self):
...         return Foo.__instance_count
...
>>> a = Foo()
>>> print('Instance count %s' % a.foo_instances())
Instance count 1
>>> b = Foo()
>>> print('Instance count %s' % a.foo_instances())
Instance count 2
>>> Foo.foo_instances()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo_instances() missing 1 required positional argument: 'self'
```

Attributes

The next idea, which still doesn't solve our problem, consists in omitting the parameter **self** and adding **@staticmethod** decorator.

```
>>> class Foo(object):
...     __instance_count = 0
...     def __init__(self):
...         type(self).__instance_count += 1
...     @staticmethod
...     def foo_instances():
...         return Foo.__instance_count
...
>>> a = Foo()
>>> print('Instance count %s' % Foo.foo_instances())
Instance count 1
>>> print('Instance count %s' % a.foo_instances())
Instance count 1
>>> b = Foo()
>>> print('Instance count %s' % Foo.foo_instances())
Instance count 2
>>> print('Instance count %s' % a.foo_instances())
Instance count 2
>>> print('Instance count %s' % b.foo_instances())
Instance count 2
```

Class Methods

@classmethod

Static methods shouldn't be confused with class methods.

Like static methods class methods are not bound to instances, but unlike static methods class methods are bound to a class.

The first parameter of a class method is a reference to a class, i.e. a class object. They can be called via an instance or the class name.

@classmethod

```
>>> class Foo(object):
...     __instance_count = 0
...     def __init__(self):
...         type(self).__instance_count += 1
...     @classmethod
...     def foo_instances(cls):
...         return cls, Foo.__instance_count
...
>>> Foo.foo_instances
<bound method Foo.foo_instances of <class '__main__.Foo'>>
>>> Foo.foo_instances()
(<class '__main__.Foo'>, 0)
>>> a = Foo()
>>> Foo.foo_instances()
(<class '__main__.Foo'>, 1)
>>> a.foo_instances()
(<class '__main__.Foo'>, 1)
>>> b = Foo()
>>> a.foo_instances()
(<class '__main__.Foo'>, 2)
>>> b.foo_instances()
(<class '__main__.Foo'>, 2)
>>> Foo.foo_instances()
(<class '__main__.Foo'>, 2)
```

@classmethod

Example with both **@staticmethod** and **@classmethod**

```
>>> class fraction(object):
...     def __init__(self, n, d):
...         self.numerator, self.denominator = fraction.reduce(n, d)
...     @staticmethod
...     def gcd(a,b):
...         while b != 0:
...             a, b = b, a%b
...         return a
...     @classmethod
...     def reduce(cls, n1, n2):
...         g = cls.gcd(n1, n2)
...         return (n1 // g, n2 // g)
...     def __str__(self):
...         return str(self.numerator)+'/'+str(self.denominator)
...
>>> x = fraction(8,24)
>>> x
<__main__.fraction object at 0x101448978>
>>> print(x)
1/3
```

@classmethod

The usefulness of class methods in inheritance.

```
>>> class BaseClass(object):
...     name = 'Base class'
...     @staticmethod
...     def about():
...         print('This is %s' % BaseClass.name)
...
>>> class SubClass1(BaseClass):
...     name = 'Sub Class 1'
...
>>> class SubClass2(BaseClass):
...     name = 'Sub Class 2'
...
>>> base_class = BaseClass()
>>> base_class.about()
This is Base class
>>> sub_class_1 = SubClass1()
>>> sub_class_1.about()
This is Base class
>>> sub_class_2 = SubClass2()
>>> sub_class_2.about()
This is Base class
```

@classmethod

Solution is @classmethod not @staticmethod

```
>>> class BaseClass(object):
...     name = 'Base class'
...     @classmethod
...     def about(cls):
...         print('This is %s' % cls.name)

>>> class SubClass1(BaseClass):
...     name = 'Sub Class 1'

>>> class SubClass2(BaseClass):
...     name = 'Sub Class 2'

>>> base_class = BaseClass()
>>> base_class.about()
This is Base class

>>> sub_class_1 = SubClass1()
>>> sub_class_1.about()
This is Sub Class 1

>>> sub_class_2 = SubClass2()
>>> sub_class_2.about()
This is Sub Class 2
```

Inheritance

Inheritance

Simple Inheritance Example

```
>>> class BaseClass(object):
...     def __init__(self, val1, val2):
...         self.val1 = val1
...         self.val2 = val2
...     def about(self):
...         return '%s-%s' % (self.val1, self.val2)

>>> class SubClass(BaseClass):
...     def __init__(self, val1, val2, val3):
...         BaseClass.__init__(self, val1, val2)
...         self.val3 = val3
...     def sub_about(self):
...         return '%s-%s' % (self.about(), self.val3)

>>> a = BaseClass(1, 2)
>>> a.about()
'1-2'
>>> b = SubClass(1, 2, 3)
>>> b.sub_about()
'1-2-3'
```

Inheritance with super

```
>>> class BaseClass(object):
...     def __init__(self, val1, val2):
...         self.val1 = val1
...         self.val2 = val2
...     def about(self):
...         return '%s-%s' % (self.val1, self.val2)

>>> class SubClass(BaseClass):
...     def __init__(self, val1, val2, val3):
...         super(SubClass, self).__init__(val1, val2)
...         self.val3 = val3
...     def sub_about(self):
...         return '%s-%s' % (self.about(), self.val3)

>>> a = BaseClass(1, 2)
>>> a.about()
'1-2'
>>> b = SubClass(1, 2, 3)
>>> b.sub_about()
'1-2-3'
```

Inheritance

Inheritance with super works for Python 3

```
>>> class BaseClass(object):
...     def __init__(self, val1, val2):
...         self.val1 = val1
...         self.val2 = val2
...     def about(self):
...         return '%s-%s' % (self.val1, self.val2)

>>> class SubClass(BaseClass):
...     def __init__(self, val1, val2, val3):
...         super().__init__(val1, val2)
...         self.val3 = val3
...     def sub_about(self):
...         return '%s-%s' % (self.about(), self.val3)

>>> a = BaseClass(1, 2)
>>> a.about()
'1-2'
>>> b = SubClass(1, 2, 3)
>>> b.sub_about()
'1-2-3'
```

Inheritance

Inheritance with super works for Python 3. Using **__str__** for the BaseClass.

```
>>> class BaseClass(object):
...     def __init__(self, val1, val2):
...         self.val1 = val1
...         self.val2 = val2
...     def __str__(self):
...         return '%s-%s' % (self.val1, self.val2)

>>> class SubClass(BaseClass):
...     def __init__(self, val1, val2, val3):
...         super().__init__(val1, val2)
...         self.val3 = val3

>>> a = BaseClass(1, 2)
>>> print(a)
1-2
>>> b = SubClass(1, 2, 3)
>>> print(b)
1-2
```

Inheritance

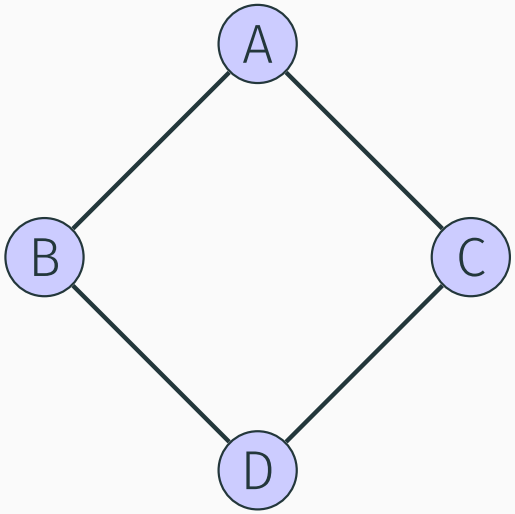
We have overridden the method **__str__** from BaseClass in SubClass

```
>>> class BaseClass(object):
...     def __init__(self, val1, val2):
...         self.val1 = val1
...         self.val2 = val2
...     def __str__(self):
...         return '%s-%s' % (self.val1, self.val2)

>>> class SubClass(BaseClass):
...     def __init__(self, val1, val2, val3):
...         super().__init__(val1, val2)
...         self.val3 = val3
...     def __str__(self):
...         return '%s-%s' % (super().__str__(), self.val3)

>>> a = BaseClass(1, 2)
>>> print(a)
1-2
>>> b = SubClass(1, 2, 3)
>>> print(b)
1-2-3
```

Diamond Problem

Multiple Inheritance

The “Diamond Problem” (sometimes referred to as the “Deadly Diamond of Death”) is the generally used term for an ambiguity that arises when two classes **B** and **C** inherit from a superclass **A**, and another class **D** inherits from both **B** and **C**. If there is a method **m** in **A** that **B** or **C** (or even both of them) has overridden, and furthermore, if does not override this method, then the question is which version of the method does **D** inherit? It could be the one from **A**, **B** or **C**.



Multiple Inheritance

```
>>> class A:
...     def m(self):
...         print("m of A called")
...
>>> class B(A):
...     def m(self):
...         print("m of B called")
...
>>> class C(A):
...     def m(self):
...         print("m of C called")
...
>>> class D(B,C):
...     pass
...
>>> x = D()
>>> x.m()
m of B called
```

Multiple Inheritance

Transpose the order of the classes in the class header of **D**, **class D(C,B):**

```
>>> class A:
...     def m(self):
...         print("m of A called")
...
>>>
>>> class B(A):
...     def m(self):
...         print("m of B called")

>>> class C(A):
...     def m(self):
...         print("m of C called")

>>> class D(C,B):
...     pass

>>> x = D()
>>> x.m()
m of C called
```

Multiple Inheritance

Result using Python2.7

```
>>> class A:
...     def m(self):
...         print("m of A called")
...
>>> class B(A):
...     pass
...
>>>
... class C(A):
...     def m(self):
...         print("m of C called")
...
>>> class D(B,C):
...     pass
...
>>> x = D()
>>> x.m()
m of A called
```

Multiple Inheritance

Result using Python3

```
>>> class A:
...     def m(self):
...         print("m of A called")

>>> class B(A):
...     pass

... class C(A):
...     def m(self):
...         print("m of C called")

>>> class D(B,C):
...     pass

>>> x = D()
>>> x.m()
m of C called
```

Multiple Inheritance

Only for those who are interested in Python version2: To have the same inheritance behavior in Python2 as in Python3, every class has to inherit from the class **object**. Our class A doesn't inherit from **object**, so we get a so-called old-style class, if we call the script with python2. Multiple inheritance with old-style classes is governed by two rules: depth-first and then left-to-right. If you change the header line of A into **class A(object):**, we will have the same behavior in both Python versions.

Multiple Inheritance

Result using Python2.7

```
>>> class A(object):
...     def m(self):
...         print("m of A called")

>>> class B(A):
...     pass

... class C(A):
...     def m(self):
...         print("m of C called")

>>> class D(B,C):
...     pass

>>> x = D()
>>> x.m()
m of C called
```

Multiple Inheritance

Result using Python3

```
>>> class A(object):
...     def m(self):
...         print("m of A called")

>>> class B(A):
...     pass

... class C(A):
...     def m(self):
...         print("m of C called")

>>> class D(B,C):
...     pass

>>> x = D()
>>> x.m()
m of C called
```

Method Resolution Order

Method Resolution Order

```
>>> class A:
...     def m(self):
...         print("m of A called")
>>> class B(A):
...     def m(self):
...         print("m of B called")
>>> class C(A):
...     def m(self):
...         print("m of C called")
>>> class D(B,C):
...     def m(self):
...         print("m of D called")
```

Let's apply the method **m** on an instance of **D**. We can see that only the code of the method **m** of **D** will be executed. We can also explicitly call the methods m of the other classes via the class name.

```
>>> x = D()
>>> B.m(x)
m of B called
>>> C.m(x)
m of C called
>>> A.m(x)
m of A called
```

Method Resolution Order

Now let's assume that the method **m** of **D** should execute the code of **m** of **B**, **C** and **A** as well, when it is called. We could implement it like this:

```
>>> class D(B,C):
...     def m(self):
...         print("m of D called")
...         B.m(self)
...         C.m(self)
...         A.m(self)

>>> x = D()
>>> x.m()
m of D called
m of B called
m of C called
m of A called
```

Method Resolution Order

But it turns out once more that things are more complicated than it seems.

How can we cope with the situation, if both **m** of **B** and **m** of **C** will have to call **m** of **A** as well?

In this case, we have to take away the call **A.m(self)** from **m** in **D**.

The code might look like this, but there is still a bug lurking in it.

Method Resolution Order

```
>>> class A:
...     def m(self):
...         print("m of A called")
...
>>> class B(A):
...     def m(self):
...         print("m of B called")
...         A.m(self)
...
>>> class C(A):
...     def m(self):
...         print("m of C called")
...         A.m(self)
...
>>> class D(B,C):
...     def m(self):
...         print("m of D called")
...         B.m(self)
...         C.m(self)
```

Method Resolution Order

The bug is that the method **m** of **A** will be called twice:

```
>>> x = D()
>>> x.m()
m of D called
m of B called
m of A called
m of C called
m of A called
```

Method Resolution Order

One way to solve this problem - admittedly not a Pythonic one - consists in splitting the methods **m** of **B** and **C** in two methods.

The first method, called **_m** consists of the specific code for **B** and **C** and the other method is still called **m**, but consists now of a call **self._m()** and a call **A.m(self)**.

The code of the method **m** of **D** consists now of the specific code of **D print("m of D called")**, and the calls **B._m(self)**, **C._m(self)** and **A.m(self)**

Method Resolution Order

Our problem is solved, but - as we have already mentioned - not in a pythonic way.

```
>>> class A:
...     def m(self):
...         print("m of A called")
...
>>> class B(A):
...     def _m(self):
...         print("m of B called")
...     def m(self):
...         self._m()
...         A.m(self)
...
>>> class C(A):
...     def _m(self):
...         print("m of C called")
...     def m(self):
...         self._m()
...         A.m(self)
...
>>> class D(B,C):
...     def m(self):
...         print("m of D called")
...         B._m(self)
...         C._m(self)
...         A.m(self)
```

Method Resolution Order

Our problem is solved, but - as we have already mentioned - not in a pythonic way.

```
>>> x = D()
>>> x.m()
m of D called
m of B called
m of C called
m of A called
```

Method Resolution Order

The optimal way to solve the problem, which is the "**super**" pythonic way, consists in calling the super function

```
>>> class A(object):
...     def m(self):
...         print("m of A called")
>>> class B(A):
...     def m(self):
...         print("m of B called")
...         super().m()
>>> class C(A):
...     def m(self):
...         print("m of C called")
...         super().m()
>>> class D(B,C):
...     def m(self):
...         print("m of D called")
...         super().m()
>>> x = D()
>>> x.m()
m of D called
m of B called
m of C called
m of A called
```

Method Resolution Order

The super function is often used when instances are initialized with the **__init__** method

```
>>> class A(object):
...     def __init__(self):
...         print("A.__init__")
>>> class B(A):
...     def __init__(self):
...         print("B.__init__")
...         super().__init__()
>>> class C(A):
...     def __init__(self):
...         print("C.__init__")
...         super().__init__()
>>> class D(B,C):
...     def __init__(self):
...         print("D.__init__")
...         super().__init__()
>>> x = D()
```

Method Resolution Order

The super function is often used when instances are initialized with the **__init__** method

```
>>> d = D()
D.__init__
B.__init__
C.__init__
A.__init__
>>> c = C()
C.__init__
A.__init__
>>> b = B()
B.__init__
A.__init__
>>> a = A()
A.__init__
```

Method Resolution Order

The question arises how the super functions makes its decision. How does it decide which class has to be used? As we have already mentioned, it uses the so-called method resolution **order(MRO)**. It is based on the “C3 superclass linearization” algorithm. This is called a linearization, because the tree structure is broken down into a linear order. The MRO method can be used to create this list

```
>>> D.mro()
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
>>> C.mro()
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
>>> B.mro()
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
>>> A.mro()
[<class '__main__.A'>, <class 'object'>]
```

Polymorphism

Polymorphism

What polymorphism means in the programming language context?
Polymorphism in Computer Science is the ability to present the same interface for differing underlying forms. We can have in some programming languages polymorphic functions or methods for example. Polymorphic functions or methods can be applied to arguments of different types, and they can behave differently depending on the type of the arguments to which they are applied. We can also define the same function name with a varying number of parameter.

Polymorphism

```
>>> def f(x, y):
...     print("values: ", x, y)
...
>>> f(42, 43)
values:  42 43
>>> f(42, 43.7)
values:  42 43.7
>>> f(42.3, 43)
values:  42.3 43
>>> f(42.0, 43.9)
values:  42.0 43.9
>>> f([3,5,6],(3,5))
values:  [3, 5, 6] (3, 5)
```

Slots

Slots

The attributes of objects are stored in a dictionary **__dict__**. Like any other dictionary, a dictionary used for attribute storage doesn't have a fixed number of elements. In other words, you can add elements to dictionaries after they have been defined.

This is the reason, we can dynamically add attributes to objects of classes that we have created.

```
>>> class Foo(object):
...     pass
...
>>> a = Foo()
>>> a.x = 128
>>> a.y = "Dynamically created Attribute!"
# The dictionary containing the attributes of "a" can be accessed
>>> a.__dict__
{'y': 'Dynamically created Attribute!', 'x': 128}
```

Slots

We can't do this with built-in classes like '**int**', or '**list**'.

```
>>> a = int()
>>> a.x = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'x'
>>> a = dict()
>>> a.y = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'y'
>>> a.list
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'list'
>>> a.x = 53
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'x'
```

Slots

Using a dictionary for attribute storage is very convenient, but it can mean a waste of space for objects, which have only a small amount of instance variables. The space consumption can become critical when creating large numbers of instances. Slots are a nice way to work around this space consumption problem. Instead of having a dynamic dict that allows adding attributes to objects dynamically, slots provide a static structure which prohibits additions after the creation of an instance.

When we design a class, we can use slots to prevent the dynamic creation of attributes. To define slots, you have to define a list with the name **__slots__**. The list has to contain all the attributes, you want to use. We demonstrate this in the following class, in which the slots list contains only the name for an attribute "**val**".

Slots

We fail to create an attribute “**new**”

```
>>> class Foo(object):
...     __slots__ = ['val']
...     def __init__(self, v):
...         self.val = v

>>> x = Foo(42)
>>> x.val
42
>>> x.new = "not possible"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Foo' object has no attribute 'new'
```

Slots

We mentioned in the beginning that slots are preventing a waste of space with objects. Since, **Python 3.3** this advantage is not as impressive any more. With **Python 3.3** and above Key-Sharing Dictionaries are used for the storage of objects. The attributes of the instances are capable of sharing part of their internal storage between each other, i.e. the part which stores the keys and their corresponding hashes. This helps to reduce the memory consumption of programs, which create many instances of non-builtin types.

Relationship between Class and type

Classes and Class Creation

The relationship between type and class. When you have defined classes so far, you may have asked yourself, what is happening "**behind the lines**'. We have already seen, that applying "**type**" to an object returns the class of which the object is an instance of:

```
>>> x = [4, 5, 9]
>>> y = "Hello"
>>> print(type(x), type(y))
<class 'list'> <class 'str'>
```

If you apply type on the name of a class itself, you get the class **type** returned.

```
>>> print(type(list), type(str))
<class 'type'> <class 'type'>
```

Classes and Class Creation

This is similar to applying type on **type(x)** and **type(y)**

```
>>> x = [4, 5, 9]
>>> y = "Hello"
>>> print(type(x), type(y))
<class 'list'> <class 'str'>
>>> print(type(type(x)), type(type(y)))
<class 'type'> <class 'type'>
```

Classes and Class Creation

A user-defined class (or class object) is an instance of the object named “**type**”, which is itself a class.

The classes are created from type.

A class is an instance of the class “**type**”. In Python3 there is no difference between “**classes**” and “**types**”. They are in most cases used as synonyms.

The fact that classes are instances of a class “**type**” allows us to program metaclasses. We can create classes, which inherit from the class “**type**”. So, a metaclass is a subclass of the class “**type**”.

Instead of only one argument, type can be called with three parameters:

```
1   type(classname, superclasses, attributesDict)
```

Classes and Class Creation

If type is called with three arguments, it will return a new type object. This provides us with a dynamic form of the class statement.

“**classname**” is a string defining the class name and becomes the name attribute; “**superclasses**” is a list or tuple with the superclasses of our class. This list or tuple will become the bases attribute; the attributesDict is a dictionary, functioning as the namespace of our class. It contains the definitions for the class body and it becomes the dict attribute.

Let’s have a look at a simple class definition:

```
>>> class Foo(object):
...     pass

>>> a = Foo()
>>> type(a)
<class '__main__.Foo'>
```

Classes and Class Creation

We can use “**type**” for the previous class definition.

```
>>> Foo = type("Foo", (), {})
>>> x = Foo()
>>> print(type(x))
<class '__main__.Foo'>
```

When we call “**type**”, the call method of type is called. The call method runs two other methods: **new** and **init**

```
type.__new__(typeclass, classname, superclasses, attributedict)
type.__init__(cls, classname, superclasses, attributedict)
```

Class creating without **type**

```
>>> class ObjectWithOutType(object):
...     counter = 0
...     def __init__(self, name):
...         self.name = name
...     def about(self):
...         return 'This is test hello!' + self.name

>>> y = ObjectWithOutType('without')
>>> print(y.name)
without
>>> print(y.about())
This is test hello!without
>>> print(y.__dict__)
{'name': 'without'}
```

Classes and Class Creation

The **new** method creates and returns the **new** class object, and after this the **new** method initializes the newly created object.

```
>>> def object_init(self, name):
...     self.name = name
>>> ObjectWithType = type('ObjectWithType',
...                       (),
...                       {'counter': 0,
...                       '__init__': object_init,
...                       'about': lambda self: 'This is from Object with type! ' + self.name})

>>> x = ObjectWithType('with')
>>> print(x.name)
with
>>> print(x.about())
This is from Object with type! with
>>> print(x.__dict__)
{'name': 'with'}
```

More metaclasses

Consider the following example.

```
>>> class BaseClass1:
...     def ret_method(self, *args):
...         return 5

>>> class BaseClass2:
...     def ret_method(self, *args):
...         return 5

>>> a = BaseClass1()
>>> a.ret_method()
5
>>> b = BaseClass2()
>>> b.ret_method()
5
```

More metaclasses

Another way to achieve this

```
>>> class SuperClass(object):
...     def ret_method(self, *args):
...         return 5

>>> class BaseClass1(SuperClass):
...     pass

>>> class BaseClass2(SuperClass):
...     pass

>>> a = BaseClass1()
>>> a.ret_method()
5
>>> b = BaseClass2()
>>> b.ret_method()
5
```

More metaclasses

Let's assume, we don't know a previously if we want or need this method. Let's assume that the decision, if the classes have to be augmented, can only be made at runtime. This decision might depend on configuration files, user input or some calculations.

```
>>> required = True
>>> def ret_method(self, *args):
...     return 5
>>> class BaseClass1(object):
...     pass
>>> if required:
...     BaseClass1.ret_method = ret_method
... class BaseClass2(object):
...     pass
>>> if required:
...     BaseClass2.ret_method = ret_method
>>> a = BaseClass1()
>>> a.ret_method()
5
>>> b = BaseClass2()
>>> b.ret_method()
5
```

More metaclasses

We have to add the same code to every class and it seems likely that we might forget it. Hard to manage and maybe even confusing, if there are many methods we want to add.

```
>>> required = True
>>> def ret_method(self, *args):
...     return 5
>>> def augment_method(cls):
...     if required:
...         cls.ret_method = ret_method
...     return cls
>>> class BaseClass1(object):
...     pass
>>> augment_method(BaseClass1)
>>> class BaseClass2(object):
...     pass
>>> augment_method(BaseClass2)
>>> a = BaseClass1()
>>> a.ret_method()
5
>>> b = BaseClass2()
>>> b.ret_method()
5
```

More metaclasses

This is good but we need to make sure the code is executed.

The code should be executed automatically. We need a way to make sure that **some** code might be executed automatically after the end of a class definition.

```
>>> required = True
>>> def ret_method(self, *args):
...     return 5
>>> def augment_method(cls):
...     if required:
...         print('Adding')
...         cls.ret_method = ret_method
...     return cls
>>> @augment_method
... class BaseClass1:
...     pass
>>> @augment_method
... class BaseClass2:
...     pass
>>> a = BaseClass1()
>>> a.ret_method()
5
>>> b = BaseClass2()
>>> b.ret_method()
5
```

Singleton

Singleton

```
>>> class Singleton(type):
...     _instances = {}
...     def __call__(cls, *args, **kwargs):
...         if cls not in cls._instances:
...             cls._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
...         return cls._instances[cls]
...
>>> class SingletonClass(metaclass=Singleton):
...     pass
...
>>> class RegularClass():
...     pass
...
>>> x = SingletonClass()
>>> y = SingletonClass()
>>> print(x == y)
True
>>> x = RegularClass()
>>> y = RegularClass()
>>> print(x == y)
False
```

Common Mistakes

Common Mistakes

```
>>> a = [1, 2, 3]
>>> b = a
>>> b.append(4)
>>> b
[1, 2, 3, 4]
>>> a
[1, 2, 3, 4]
```

Common Mistakes

```
>>> def foo(a=[]):
...     a.append(1)
...     return a
...
>>> foo()
[1]
>>> foo()
[1, 1]
>>> foo()
[1, 1, 1]
```

Common Mistakes

```
>>> import time
>>> def foo(a={}):
...     a.update({1: time.time()})
...     return a
...
>>> a = foo()
>>> a
{1: 1476786062.803211}
>>> b = foo()
>>> b
{1: 1476786068.338572}
>>> a
{1: 1476786068.338572}
```

Common Mistakes

Best practice is to define **None**

Recursion

Recursion

Recursion is a logical relationship between problems of essentially reduced versions of the original problem.

Every recursive procedure has a logical reading. Recursive procedure calls itself either directly or indirectly.

```
1  >>> def fact(n):
2  ...     if n == 0: return 1
3  ...     else: return n * fact(n-1)
```

“When the recursion appears at the end of a simple function, it’s described as a tail call optimization. Many compilers will optimize this into a loop. Python lacking this optimization in it’s compiler doesn’t do this kind of tail-call transformation.” - from a book

A lot of FUDD (**Fear Uncertainty Doubt Disinformation**) among even book authors.

Disinformation about Tail Recursion Optimization

Schemers, Prologgers and Little Lispers denounce Python for not having Tail Recursion Optimization

Recursion

```
>>> def factorial(n):
...     return n * factorial(n-1) if n > 1 else 1
>>> factorial(5)
120
>>> def tail_factorial(n, acc=1):
...     return tail_factorial(n-1, n*acc) if n > 1 else acc
>>> tail_factorial(5)
120
```

Recursion

Exercise: How to sum the Integers from 1 to N?

Using recursion and tail recursion.

Recursion

```
>>> def sum(n):
...     return n + sum(n-1) if n > 0 else 0
>>> sum(5)
15
>>> def tail_sum(n, acc=0):
...     return tail_sum(n-1, n+acc) if n > 0 else acc
>>> tail_sum(5)
15
```

Recursion

Binary Tree Binary tree is one where each internal node has a maximum of two children.

```
>>> class Node(object):
...     def __init__(self, data, left=None, right=None):
...         self.data = data
...         self.left = left
...         self.right = right
```

Recursion

Time for exercise on Binary Tree

- Return the count of nodes of a binary tree.
- Return the maximum of data fields of nodes.
- Return the height of the tree.
- Return the number of leaves in the tree.

Recursion

Binary Tree Return the count of nodes of a binary tree.

```
>>> def sizeof(node):
...     '''
...     >>> t = Node(24)
...     >>> t.left = Node(7, Node(10), Node(18))
...     >>> t.right = Node(9)
...     >>> sizeof(t)
...     5
...     >>> sizeof(Node(2))
...     1
...     '''
...     return sizeof(node.left) + 1 + sizeof(node.right) if node else 0
```

Binary Tree Return the maximum of data fields of nodes.

```
>>> def findMaxNodeValue(node):
...     '''
...     >>> t = Node(-24)
...     >>> t.left = Node(-7, Node(-10), Node(-18))
...     >>> t.right = Node(-9)
...     >>> findMaxNodeValue(t)
...     -7
...     >>> findMaxNodeValue(Node(-1))
...     -1
...     >>> t.right.right = Node(1)
...     >>> findMaxNodeValue(t)
...     1
...     >>> t.right.right.right = Node(2)
...     >>> findMaxNodeValue(t)
...     2
...     >>> t.right.right.right.right = Node(3)
...     >>> findMaxNodeValue(t)
...     3
...     '''
...     return max(findMaxNodeValue(node.left),
...                node.data,
...                findMaxNodeValue(node.right)) if node else -float('inf')
...
```

Recursion

Binary Tree Return the height of the tree.

```
>>> def height(node):
...     '''
...     >>> t = Node(24)
...     >>> t.left = Node(7, Node(10), Node(18))
...     >>> t.right = Node(9)
...     >>> height(t)
...     3
...     >>> height(Node(20))
...     1
...     >>> t.right.right = Node(1)
...     >>> t.right.right.right = Node(2)
...     >>> t.right.right.right.right = Node(3)
...     >>> height(t)
...     5
...     '''
...     return 1 + max(height(node.left), height(node.right)) if node else 0
...
>>> doctest.run_docstring_examples(height, globals(), True)
```

Recursion

Binary Tree Return the number of leaves in the tree.

```
>>> def numberOfLeaves(node):
...     '''
...     >>> t = Node(24)
...     >>> t.left = Node(7, Node(10), Node(18))
...     >>> t.right = Node(9)
...     >>> numberOfLeaves(t)
...     3
...     >>> numberOfLeaves(Node(20))
...     1
...     '''
...     return (numberOfLeaves(node.left) +
...             numberOfLeaves(node.right)) if node else 0
...
>>> doctest.run_docstring_examples(numberOfLeaves, globals(), True)
```

Python 3.x

Python 3.x

```
>>> a, b = range(2)
>>> a, b
(0, 1)
```

Python 3.x

```
>>> a, b = range(2)
>>> a, b
(0, 1)
>>> a, b, *rest = range(10)
>>> a, b, rest
(0, 1, [2, 3, 4, 5, 6, 7, 8, 9])
```

***rest** can go anywhere

```
>>> a, b = range(2)
>>> a, b
(0, 1)
>>> a, b, *rest = range(10)
>>> a, b, rest
(0, 1, [2, 3, 4, 5, 6, 7, 8, 9])
>>>
>>>
>>> a, *rest, b = range(10)
>>> a, rest, b
(0, [1, 2, 3, 4, 5, 6, 7, 8], 9)
```

Opening the following sample.txt file using python. And to read only the first and last lines.

```
This is line 1
This is line 2
This is line 3
This is line 4
This is line 5
This is line 6
This is line 7
This is line 8
```

```
>>> with open('sample.txt') as f:
...     first, *_, last = f.readlines()
...
>>> first
'This is line 1\n'
>>> last
'This is line 8\n'
```

Refactoring methods

```
>>> def foo(a, b, *args):
...     pass
```

TO

```
>>> def foo(*args):
...     a, b, *args = args
...     pass
```

```
>>> def f(a, b, *args, option=True):
...     pass
```

- option comes *after* ***args**.
- The only way to access it is to explicitly call **f(a, b, option=True)**
- You can write just a ***** if you don't want to collect ***args**.

```
>>> def f(a, b, *, option=True):
...     pass
```

Refactoring methods

```
>>> 'abc' > 123
True
>>> None > all
False
```

TO

```
>>> 'one' > 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() > int()
```

Python 3.x yield

Refactoring methods

```
for i in gen():
    yield i
```

TO

```
yield from gen()
```

I reordered the keyword arguments of a function, but something was implicitly passing in arguments expecting the order.

```
>>> def maxall(iterable, key=None):
...     """
...     A list of all max items from the iterable
...     """
...     key = key or (lambda x: x)
...     m = max(iterable, key=key)
...     return [i for i in iterable if key(i) == key(m)]

>>> maxall(['a', 'ab', 'bc'], len)
['ab', 'bc']
```

The max builtin supports **max(a, b, c)**. We should allow that too.

```
>>> def maxall(*args, key=None):
...     """
...     A list of all max items from the iterable
...     """
...     if len(args) == 1:
...         iterable = args[0]
...     else:
...         iterable = args
...     key = key or (lambda x: x)
...     m = max(iterable, key=key)
...     return [i for i in iterable if key(i) == key(m)]
```

We just broke any code that passed in the key as a second argument without using the keyword.

```
>>> maxall(['a', 'ab', 'ac'], len)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 10, in maxall
TypeError: unorderable types: builtin_function_or_method() > list()
```

Actually in Python 2 it would just return **['a', 'ab', 'ac']**

By the way, **max** shows that this is already possible in Python 2, but only if you write your function in **C**.

Obviously, we should have used **maxall(iterable, *, key=None)** to begin with.

Python 3.x yield

Bad

```
def dup(n):
  A = []
  for i in range(n):
      A.extend([i, i])
  return A
```

Good

```
def dup(n):
  for i in range(n):
      yield i
      yield i
```

Better

```
def dup(n):
  for i in range(n):
      yield from [i, i]
```

Namedtuple

namedtuple

Factory Function for Tuples with Named Fields

```
>>> from collections import namedtuple
>>> import csv
>>> Point = namedtuple('Point', 'x y')
>>> for point in map(Point._make, csv.reader(open("points.csv", "r"))):
...     print(point.x, point.y)
1 2
2 3
3.4 5.6
10 10
>>> p = Point(x=11, y=22)
>>> p._asdict()
OrderedDict([('x', 11), ('y', 22)])
>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

Factory Function for Tuples with Named Fields

```
>>> from collections import namedtuple
>>> RandomClass = namedtuple('RandomClass', 'class random')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/lib/python3.5/collections/__init__.py", line 403, in namedtuple 'keyword: %r' % name)
ValueError: Type names and field names cannot be a keyword: 'class'
>>> RandomClass = namedtuple('RandomClass', 'class random', rename=True)
>>> RandomClass._fields
('_0', 'random')
>>> DuplicateClass = namedtuple('DuplicateClass', 'abc xyz abc xyz')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "~/lib/python3.5/collections/__init__.py", line 410, in namedtuple
    raise ValueError('Encountered duplicate field name: %r' % name)
ValueError: Encountered duplicate field name: 'abc'
>>> DuplicateClass = namedtuple('DuplicateClass', 'abc xyz abc xyz', rename=True)
>>> DuplicateClass._fields
('abc', 'xyz', '_2', '_3')
```

Signal Handler

Signal Handler

Simple Alarm example

```
>>> import signal
>>> import time
>>>
>>> def receive_alarm(signum, stack):
...     print('Alarm :', time.ctime())

>>> # Call receive_alarm in 2 seconds
... signal.signal(signal.SIGALRM, receive_alarm)
<Handlers.SIG_DFL: 0>
>>> signal.alarm(2)
0
>>>
>>> print('Before:', time.ctime())
Before: Tue Oct 18 14:10:00 2016
>>> time.sleep(4)
Alarm : Tue Oct 18 14:10:02 2016
>>> print('After :', time.ctime())
After : Tue Oct 18 14:10:04 2016
```

Conclusion

References

- https://docs.python.org/3.6/
- http://www.python-course.eu/
- How to Make Mistakes in Python
- Functional Programming in Python
- Picking a Python Version: A Manifesto
- asmeurer

Thank you!

Questions?