Python Tips, Tricks, and Hidden Features

(using type, collections, meta-classes, and more)


I'm Dustin
http://github.com/di


promptworks

Irked.

jeffdeville 3:40 PM

In python, how can I create an empty object, and just start assigning properties to it. I don't want to have to define a class that is empty just to do this.

```
j = object()
j.hi = "there"
```

doesn't work... I'm irked

Irked.

```
>>> j = object()
>>> j.hi = 'there'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'object' object has no attribute 'hi'
```

type

type

```
>>> type(['foo', 'bar'])
```

type

```
>>> type(['foo', 'bar'])
<type 'list'>
```

type

```
>>> type(list)
```

type

```
>>> type(list)
<type 'type'>
```

type

```
>>> type(type)
```

type

```
>>> type(type)
<type 'type'>
```

type

```
>>> type(None)
```

type

```
>>> type(None)
<type 'NoneType'>
```

type

```
>>> def func():
...     pass
...
>>> type(func)
```

type

```
>>> def func():
...     pass
...
>>> type(func)
<type 'function'>
```

type

```
>>> import types
>>> dir(types)
['BooleanType', 'BufferType', 'BuiltinFunctionType', 'BuiltinMethodType',
'ClassType', 'CodeType', 'ComplexType', 'DictProxyType', 'DictType',
'DictionaryType', 'EllipsisType', 'FileType', 'FloatType', 'FrameType',
'FunctionType', 'GeneratorType', 'GetSetDescriptorType', 'InstanceType',
'IntType', 'LambdaType', 'ListType', 'LongType', 'MemberDescriptorType',
'MethodType', 'ModuleType', 'NoneType', 'NotImplementedType', 'ObjectType',
'SliceType', 'StringType', 'StringTypes', 'TracebackType', 'TupleType',
'TypeType', 'UnboundMethodType', 'UnicodeType', 'XRangeType', '__all__',
'__builtins__', '__doc__', '__file__', '__name__', '__package__']
```

type

```
>>> import types
>>> type(types)
```

type

```
>>> import types
>>> type(types)
<type 'module'>
>>> type(types) == types.ModuleType
True
```

type

```
>>> class FooClass:
...     pass
...
>>> type(FooClass)
```

type

```
>>> class FooClass: # Python 2.7
...     pass
...
>>> type(FooClass)
<type 'classobj'>
```

type

```
>>> class FooClass: # Python 3.4
...     pass
...
>>> type(FooClass)
<class 'type'>
```

type

```
>>> class FooClass(object): # Python 2.7
...     pass
...
>>> type(FooClass)
<type 'type'>
```

type

```
>>> type(42) is int
True
>>> type(42)()
0
>>> int()
0
```

type

```
>>> j = type()
```

type

```
>>> j = type()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: type() takes 1 or 3 arguments
```

type

```
>>> j = type('FooClass', (object,), {'hi': 'there'})
>>> type(j)
<class 'type'>
```

type

```
>>> j = type('FooClass', (object,), {'hi': 'there'})
>>> j.hi
'there'
```

type

```
>>> j = type('', (object,), {'hi': 'there'})
>>> j.hi
'there'
```

type

```
>>> j = type('', (), {'hi': 'there'})
>>> j.hi
'there'
```

type

```
>>> j = type('', (), {})
>>> j.hi = 'there'
>>> j.hi
'there'
```

type

```
>>> j = object()
>>> j.hi = 'there'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'object' object has no attribute 'hi'
```

type

```
>>> class FooClass():
...     pass
...
>>> hasattr(FooClass(), '__dict__')
True
>>> hasattr(object(), '__dict__')
False
```

type

```
>>> from stackoverflow import getsize[1]
>>> getsize(object())
16
>>> getsize(0)
24
>>> getsize(dict())
280
>>> getsize(FooClass())
344
```

---

[1] http://stackoverflow.com/a/30316760/4842627

Questions?

collections

collections

```
>>> def group(names):
...
```

```
>>> group(['tom', 'dick', 'harry', 'guido'])
{3: ['tom'], 4: ['dick'], 5: ['harry', 'guido']}
```

collections

```
>>> def group(names):
...     d = {}
...
```

```
>>> group(['tom', 'dick', 'harry', 'guido'])
{3: ['tom'], 4: ['dick'], 5: ['harry', 'guido']}
```

collections

```
>>> def group(names):
...     d = {}
...     for name in names:
...
```

```
>>> group(['tom', 'dick', 'harry', 'guido'])
{3: ['tom'], 4: ['dick'], 5: ['harry', 'guido']}
```

collections

```
>>> def group(names):
...     d = {}
...     for name in names:
...         key = len(name)
...
```

```
>>> group(['tom', 'dick', 'harry', 'guido'])
{3: ['tom'], 4: ['dick'], 5: ['harry', 'guido']}
```

collections

```
>>> def group(names):
...     d = {}
...     for name in names:
...         key = len(name)
...         if key not in d:
...             d[key] = []
...
```

```
>>> group(['tom', 'dick', 'harry', 'guido'])
{3: ['tom'], 4: ['dick'], 5: ['harry', 'guido']}
```

collections

```
>>> def group(names):
...     d = {}
...     for name in names:
...         key = len(name)
...         if key not in d:
...             d[key] = []
...         d[key].append(name)
...

>>> group(['tom', 'dick', 'harry', 'guido'])
{3: ['tom'], 4: ['dick'], 5: ['harry', 'guido']}
```

collections

```
>>> def group(names):
...     d = {}
...     for name in names:
...         key = len(name)
...         if key not in d:
...             d[key] = []
...         d[key].append(name)
...     return d
...
>>> group(['tom', 'dick', 'harry', 'guido'])
{3: ['tom'], 4: ['dick'], 5: ['harry', 'guido']}
```

collections

```
>>> def group(names):
...     d = {}
...     for name in names:
...         key = len(name)
...         d.setdefault(key, []).append(name)
...     return d
...
>>> group(['tom', 'dick', 'harry', 'guido'])
{3: ['tom'], 4: ['dick'], 5: ['harry', 'guido']}
```

collections

```
>>> def group(names):
...     return {
...         length: [
...             name for name in names if len(name) == length
...         ]
...         for length in {len(name) for name in names}
...     }
...
>>> group(['tom', 'dick', 'harry', 'guido'])
{3: ['tom'], 4: ['dick'], 5: ['harry', 'guido']}
```

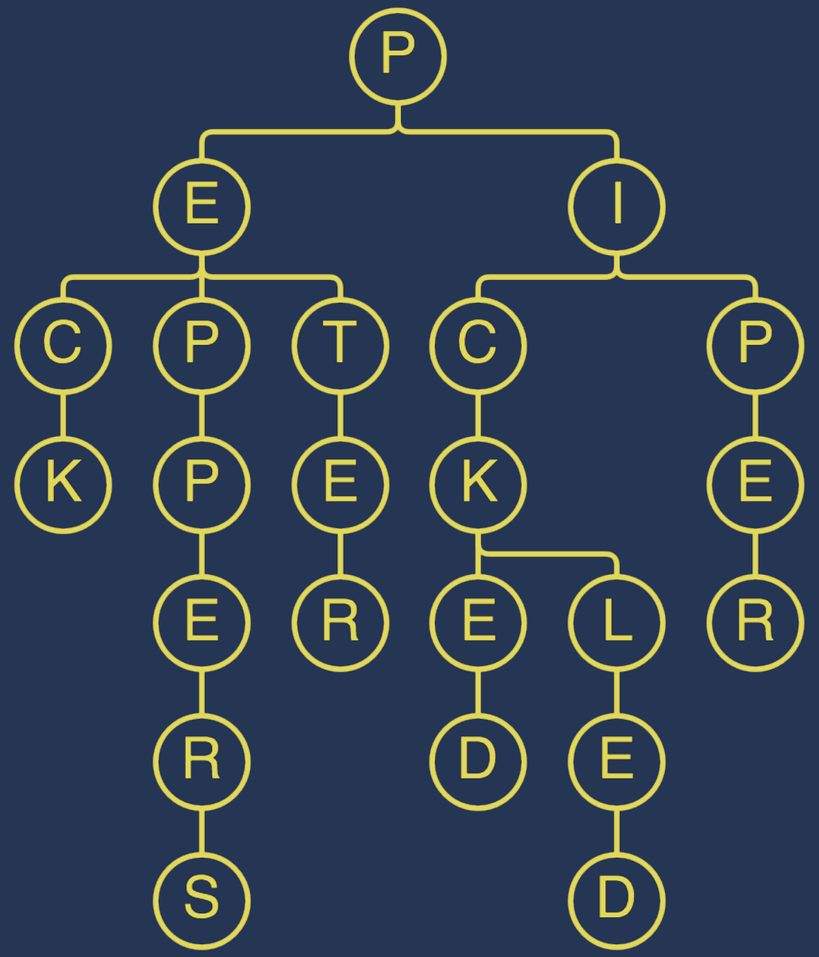
collections.defaultdict

```
>>> from collections import defaultdict
>>> def group(names):
...     d = defaultdict(list)
...     for name in names:
...         key = len(name)
...         d[key].append(name)
...     return dict(d)
...
>>> group(['tom', 'dick', 'harry', 'guido'])
{3: ['tom'], 4: ['dick'], 5: ['harry', 'guido']}
```

`collections.defaultdict`

trie: an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings[2].

e.g.: "**peter piper picked**
a **peck** of **pickled peppers**"



[2] https://en.wikipedia.org/wiki/Trie

collections.defaultdict

```
>>> trie = ...
>>> trie['b']['a']['r'] = True
>>> trie['b']['a']['r'] == True
True
>>> trie['f']['o']['o'] == True
False
>>> trie
{'b': {'a': {'r': True}}}
```

collections.defaultdict

```
>>> trie = {}
>>> trie['b']['a']['r'] = True
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'b'
```

collections.defaultdict

```
>>> from collections import defaultdict
>>> trie = defaultdict(lambda: defaultdict(dict))
>>> trie['b']['a']['r'] = True
>>> trie['b']['a']['r'] == True
True
```

collections.defaultdict

```
>>> from collections import defaultdict
>>> trie = defaultdict(lambda: defaultdict(dict))
>>> trie['b']['a']['r'] = True
>>> trie['b']['a']['r'] == True
True
>>> trie['f']['o']['o'] == True
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'o'
```

collections.defaultdict

```
>>> from collections import defaultdict
>>> trie = defaultdict(lambda: defaultdict(dict))
>>> trie['t']['r']['i']['e'] = True
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'i'
```

collections.defaultdict

```
>>> from collections import defaultdict
>>> trie = defaultdict(lambda: defaultdict(
...     lambda: defaultdict(lambda: defaultdict(
...         lambda: defaultdict(lambda: defaultdict(
...             dict))))))
```

collections.defaultdict

```
>>> from collections import defaultdict
>>> infinitedict = lambda: defaultdict(infinitedict)
>>> trie = infinitedict()
>>> trie['b']['a']['r'] = True
>>> trie['b']['a']['r'] == True
True
>>> trie['f']['o']['o'] == True
False
```

collections.defaultdict

```
>>> from collections import defaultdict
>>> def infinitedict():
...     return defaultdict(infinitedict)
...
>>> trie = infinitedict()
>>> trie['b']['a']['r'] = True
>>> trie['b']['a']['r'] == True
True
>>> trie['f']['o']['o'] == True
False
```

collections

```
>>> def count(names):
...
```

```
>>> count(['beetlejuice', 'beetlejuice', 'guido'])
{'beetlejuice': 2, 'guido': 1}
```

collections

```
>>> from collections import defaultdict
>>> def count(names):
...     d = defaultdict(int)
...
```

```
>>> count(['beetlejuice', 'beetlejuice', 'guido'])
{'beetlejuice': 2, 'guido': 1}
```

collections

```
>>> from collections import defaultdict
>>> def count(names):
...     d = defaultdict(int)
...     for name in names:
...         d[name] += 1
...

>>> count(['beetlejuice', 'beetlejuice', 'guido'])
{'beetlejuice': 2, 'guido': 1}
```

collections

```
>>> from collections import defaultdict
>>> def count(names):
...     d = defaultdict(int)
...     for name in names:
...         d[name] += 1
...     return dict(d)
...
>>> count(['beetlejuice', 'beetlejuice', 'guido'])
{'beetlejuice': 2, 'guido': 1}
```

collections.Counter

```
>>> from collections import Counter
>>> def count(names):
...     return dict(Counter(names))
...
>>> count(['beetlejuice', 'beetlejuice', 'guido'])
{'beetlejuice': 2, 'guido': 1}
```

Questions?

Metaclasses

Metaclasses

```
>>> class FooClass():
...     pass
...
```

Metaclasses

```
>>> class FooClass():
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
False
```

Metaclasses

```
>>> class Singleton():
...
```

```
>>> class FooClass(Singleton):
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
True
```

Metaclasses

```
>>> class Singleton():
...     _instance = None
...
```

```
>>> class FooClass(Singleton):
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
True
```

Metaclasses

```
>>> class Singleton():
...     _instance = None
...     def __new__(cls, *args, **kwargs):
...
```

```
>>> class FooClass(Singleton):
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
True
```

Metaclasses

```
>>> class Singleton():
...     _instance = None
...     def __new__(cls, *args, **kwargs):
...         if not cls._instance:
...


>>> class FooClass(Singleton):
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
True
```

Metaclasses

```
>>> class Singleton():
...     _instance = None
...     def __new__(cls, *args, **kwargs):
...         if not cls._instance:
...             cls._instance = object.__new__(cls, *args, **kwargs)
...

>>> class FooClass(Singleton):
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
True
```

Metaclasses

```
>>> class Singleton():
...     _instance = None
...     def __new__(cls, *args, **kwargs):
...         if not cls._instance:
...             cls._instance = object.__new__(cls, *args, **kwargs)
...         return cls._instance
...
>>> class FooClass(Singleton):
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
True
```

Metaclasses

classes : instances : : metaclasses : classes

Metaclasses

```
>>> j = type('FooClass', (object,), {'hi': 'there'})
>>> type(j)
<class 'type'>
```

Metaclasses

```
>>> j = type('FooClass', (object,), {'hi': 'there'})
>>> type(j)
<class 'type'>

>>> type(j())
<class '__main__.FooClass'>
```

Metaclasses

```
>>> class MyMeta(type):
...     pass
...
```

Metaclasses

```
>>> class MyMeta(type):
...     pass
...
>>> class FooClass(metaclass=MyMeta):
...     pass
...
```

Metaclasses

```
>>> class MyMeta(type):
...     pass
...
>>> class FooClass(object): # Python 2.7
...     __metaclass__ = MyMeta
...
```

Metaclasses

```
>>> class MyMeta(type):
...     def __new__(meta, name, bases, attrs):
...         return super().__new__(meta, name, bases, attrs)
...
>>> class FooClass(metaclass=MyMeta):
...     pass
...
```

Metaclasses

```
>>> class MyMeta(type):
...     def __new__(meta, name, bases, attrs):
...         print('New {}'.format(name))
...         return super().__new__(meta, name, bases, attrs)
...
>>> class FooClass(metaclass=MyMeta):
...     pass
...
New FooClass
```

Metaclasses

```
>>> class MyMeta(type):
...     def __call__(cls, *args, **kwargs):
...         print('Call {}'.format(cls.__name__))
...         return super().__call__(*args, **kwargs)
...
>>> class FooClass(metaclass=MyMeta):
...     pass
...
>>> f = FooClass()
Call FooClass
```

Metaclasses

```
>>> class Singleton(type):
...
```

```
>>> class FooClass(metaclass=Singleton):
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
True
```

Metaclasses

```
>>> class Singleton(type):
...     def __new__(meta, name, bases, attrs):
...
```

```
>>> class FooClass(metaclass=Singleton):
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
True
```

Metaclasses

```
>>> class Singleton(type):
...     def __new__(meta, name, bases, attrs):
...         attrs['_instance'] = None
...
```

```
>>> class FooClass(metaclass=Singleton):
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
True
```

Metaclasses

```
>>> class Singleton(type):
...     def __new__(meta, name, bases, attrs):
...         attrs['_instance'] = None
...         return super().__new__(meta, name, bases, attrs)
...
```

```
>>> class FooClass(metaclass=Singleton):
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
True
```

Metaclasses

```
>>> class Singleton(type):
...     def __new__(meta, name, bases, attrs):
...         attrs['_instance'] = None
...         return super().__new__(meta, name, bases, attrs)
...     def __call__(cls, *args, **kwargs):
...
```

```
>>> class FooClass(metaclass=Singleton):
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
True
```

Metaclasses

```
>>> class Singleton(type):
...     def __new__(meta, name, bases, attrs):
...         attrs['_instance'] = None
...         return super().__new__(meta, name, bases, attrs)
...     def __call__(cls, *args, **kwargs):
...         if not cls._instance:
...
```

```
>>> class FooClass(metaclass=Singleton):
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
True
```

Metaclasses

```
>>> class Singleton(type):
...     def __new__(meta, name, bases, attrs):
...         attrs['_instance'] = None
...         return super().__new__(meta, name, bases, attrs)
...     def __call__(cls, *args, **kwargs):
...         if not cls._instance:
...             cls._instance = super().__call__(*args, **kwargs)
...

>>> class FooClass(metaclass=Singleton):
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
True
```

Metaclasses

```
>>> class Singleton(type):
...     def __new__(meta, name, bases, attrs):
...         attrs['_instance'] = None
...         return super().__new__(meta, name, bases, attrs)
...     def __call__(cls, *args, **kwargs):
...         if not cls._instance:
...             cls._instance = super().__call__(*args, **kwargs)
...         return cls._instance
...
>>> class FooClass(metaclass=Singleton):
...     pass
...
>>> a, b = FooClass(), FooClass()
>>> a is b
True
```

Questions?

wat

wat #0

```
>>> x = ...
>>> x == x
False
```

wat #0 - Possible!

```
>>> x = float('nan')
>>> x == x
False
```

wat #0 - Possible!

```
>>> x = 0*1e400
>>> x == x
False
```

http://python-wats.herokuapp.com/

wat #1

```
>>> x = ...
>>> a = ...
>>> b = ...
>>> c = ...
>>> max(x) < max(x[a:b:c])
True
```

wat #1 - Not Possible

```
>>> x = ...
>>> a = ...
>>> b = ...
>>> c = ...
>>> max(x) < max(x[a:b:c])
True
```

wat #2

```
>>> x = ...
>>> y = ...
>>> min(x, y) == min(y, x)
False
```

wat #2 - Possible!

```
>>> x = {0}
>>> y = {1}
>>> min(x, y) == min(y, x)
False
```

wat #2 - Possible!

```
>>> min({0}, {1})
set([0])
>>> min({1}, {0})
set([1])
```

wat #2 - Possible!

```
>>> min({0}, {1})
set([0])
>>> min({1}, {0})
set([1])
>>> min({0, 1}, {0})
set([0])
```

wat #2 - Possible!

```
>>> def min(*args):
...
```

wat #2 - Possible!

```
>>> def min(*args):
...     has_item = False
...     min_item = None
...
```

wat #2 - Possible!

```
>>> def min(*args):
...     has_item = False
...     min_item = None
...     for x in args:
...
```

wat #2 - Possible!

```
>>> def min(*args):
...     has_item = False
...     min_item = None
...     for x in args:
...         if not has_item or x < min_item:
...
```

wat #2 - Possible!

```
>>> def min(*args):
...     has_item = False
...     min_item = None
...     for x in args:
...         if not has_item or x < min_item:
...             has_item = True
...             min_item = x
...
```

wat #2 - Possible!

```
>>> def min(*args):
...     has_item = False
...     min_item = None
...     for x in args:
...         if not has_item or x < min_item:
...             has_item = True
...             min_item = x
...     return min_item
...
```

wat #2 - Possible!

```
>>> {1} < {0}
False
>>> {1} < {0, 1}
True
```

wat #2 - Possible!

```
>>> {1} < {0}
False
>>> {1} < {0, 1}
True
>>> min({0}, {1})
set([0])
```

wat #3

```
>>> x = ...
>>> y = ...
>>> any(x) and not any(x + y)
True
```

wat #3 - Not Possible

```
>>> x = ...
>>> y = ...
>>> any(x) and not any(x + y)
True
```

wat #4

```
>>> x = ...
>>> y = ...
>>> x.count(y) > len(x)
True
```

wat #4 - Possible!

```
>>> x = 'a'
>>> y = ''
>>> x.count(y) > len(x)
True
```

wat #4 - Possible!

```
>>> x = 'a'
>>> y = ''
>>> x.count(y) > len(x)
True
>>> len('a')
1
```

wat #4 - Possible!

```
>>> x = 'a'
>>> y = ''
>>> x.count(y) > len(x)
True
>>> len('a')
1
>>> 'a'.count('')
2
```

wat #4 - Possible!

```
>>> def count(s, sub):
...
```

wat #4 - Possible!

```
>>> def count(s, sub):
...         result = 0
...
```

wat #4 - Possible!

```
>>> def count(s, sub):
...     result = 0
...     for i in range(len(s) + 1 - len(sub)):
...
```

wat #4 - Possible!

```
>>> def count(s, sub):
...     result = 0
...     for i in range(len(s) + 1 - len(sub)):
...         possible_match = s[i:i + len(sub)]
...
```

wat #4 - Possible!

```
>>> def count(s, sub):
...     result = 0
...     for i in range(len(s) + 1 - len(sub)):
...         possible_match = s[i:i + len(sub)]
...         if possible_match == sub:
...             result += 1
...     return result
...
```

wat #4 - Possible!

```
>>> def count(s, sub):
...     result = 0
...     for i in range(len(s) + 1 - len(sub)):
...         possible_match = s[i:i + len(sub)]
...         if possible_match == sub:
...             result += 1
...     return result
...
>>> count('a', '')
2
```

wat #4 - Possible!

```
>>> count('foofoof', 'foof') # my implementation
2
>>> 'foofoof'.count('foof')
1
```

wat #5

```
>>> x = ...
>>> y = ...
>>> z = ...
>>> x * (y * z) == (x * y) * z
False
```

wat #5 - Possible!

```
>>> x = [0]
>>> y = -1
>>> z = -1
>>> x * (y * z) == (x * y) * z
False
```

wat #5 - Possible!

```
>>> x = [0]
>>> y = -1
>>> z = -1
>>> x * (y * z) == (x * y) * z
False
>>> x * (y * z) == [0]*(-1*-1)
```

wat #5 - Possible!

```
>>> x = [0]
>>> y = -1
>>> z = -1
>>> x * (y * z) == (x * y) * z
False
>>> x * (y * z) == [0]*(-1*-1) == [0]*1
```

wat #5 - Possible!

```
>>> x = [0]
>>> y = -1
>>> z = -1
>>> x * (y * z) == (x * y) * z
False
>>> x * (y * z) == [0]*(-1*-1) == [0]*1 == [0]
True
```

wat #5 - Possible!

```
>>> x = [0]
>>> y = -1
>>> z = -1
>>> x * (y * z) == (x * y) * z
False
>>> x * (y * z) == [0]*(-1*-1) == [0]*1 == [0]
True
>>> (x * y) * z == ([0]*-1)*-1
```

wat #5 - Possible!

```
>>> x = [0]
>>> y = -1
>>> z = -1
>>> x * (y * z) == (x * y) * z
False
>>> x * (y * z) == [0]*(-1*-1) == [0]*1 == [0]
True
>>> (x * y) * z == ([0]*-1)*-1 == []*-1
```

wat #5 - Possible!

```
>>> x = [0]
>>> y = -1
>>> z = -1
>>> x * (y * z) == (x * y) * z
False
>>> x * (y * z) == [0]*(-1*-1) == [0]*1 == [0]
True
>>> (x * y) * z == ([0]*-1)*-1 == []*-1 == []
True
```

wat #6

```
>>> x = ...
>>> y = ...
>>> x < y and all(a >= b for a, b in zip(x, y))
True
```

wat #6 - Possible!

```
>>> x = []
>>> y = [0]
>>> x < y and all(a >= b for a, b in zip(x, y))
True
```

wat #6 - Possible!

```
>>> x = []
>>> y = [0]
>>> x < y and all(a >= b for a, b in zip(x, y))
True
>>> [] < [0]
True
```

wat #6 - Possible!

```
>>> x = []
>>> y = [0]
>>> x < y and all(a >= b for a, b in zip(x, y))
True
>>> [] < [0]
True
>>> zip([], [0])
[]
```

wat #6 - Possible!

```
>>> x = []
>>> y = [0]
>>> x < y and all(a >= b for a, b in zip(x, y))
True
>>> [] < [0]
True
>>> zip([], [0])
[]
>>> all([])
True
```

wat #7

```
>>> x = ...
>>> len(set(list(x))) == len(list(set(x)))
False
```

wat #7 - Not Possible

```
>>> x = ...
>>> len(set(list(x))) == len(list(set(x)))
False
```

wat #8

```
>>> x = ...
>>> min(x) == min(*x)
False
```

wat #8 - Possible!

```
>>> x = [[0]]
>>> min(x) == min(*x)
False
```

wat #8 - Possible!

```
>>> x = [[0]]
>>> min(x) == min(*x)
False
>>> min([1, 2, 3]) == min(*[1, 2, 3]) == min(1, 2, 3)
True
```

wat #8 - Possible!

```
>>> x = [[0]]
>>> min(x) == min(*x)
False
>>> min([1, 2, 3]) == min(*[1, 2, 3]) == min(1, 2, 3)
True
>>> min(x) == [0]
True
```

wat #8 - Possible!

```
>>> x = [[0]]
>>> min(x) == min(*x)
False
>>> min([1, 2, 3]) == min(*[1, 2, 3]) == min(1, 2, 3)
True
>>> min(x) == [0]
True
>>> min(*x) == min([0]) == 0
True
```

wat #9

```
>>> x = ...
>>> y = ...
>>> sum(0 * x, y) == y
False
```

wat #9 - Not Possible

```
>>> x = ...
>>> y = ...
>>> sum(0 * x, y) == y
False
```

wat #9 - Not Possible

```
>>> x = ...
>>> y = ...
>>> sum(0 * x, y) == y
False
>>> sum([1, 1, 1], 7)
10
```

wat #9 - Not Possible

```
>>> x = ...
>>> y = ...
>>> sum(0 * x, y) == y
False
>>> sum([1, 1, 1], 7)
10
>>> sum([], 7)
7
```

wat #10

```
>>> x = ...
>>> y = ...
>>> y > max(x) and y in x
True
```

wat #10 - Possible!

```
>>> x = 'aa'
>>> y = 'aa'
>>> y > max(x) and y in x
True
```

wat #10 - Possible!

```
>>> x = 'aa'
>>> y = 'aa'
>>> y > max(x) and y in x
True
>>> max('aa')
'a'
```

wat #10 - Possible!

```
>>> x = 'aa'
>>> y = 'aa'
>>> y > max(x) and y in x
True
>>> max('aa')
'a'
>>> 'aa' > 'a'
True
```

wat #10 - Possible!

```
>>> x = 'aa'
>>> y = 'aa'
>>> y > max(x) and y in x
True
>>> max('aa')
'a'
>>> 'aa' > 'a'
True
>>> 'aa' in 'aa'
True
```

Questions?

Thanks!