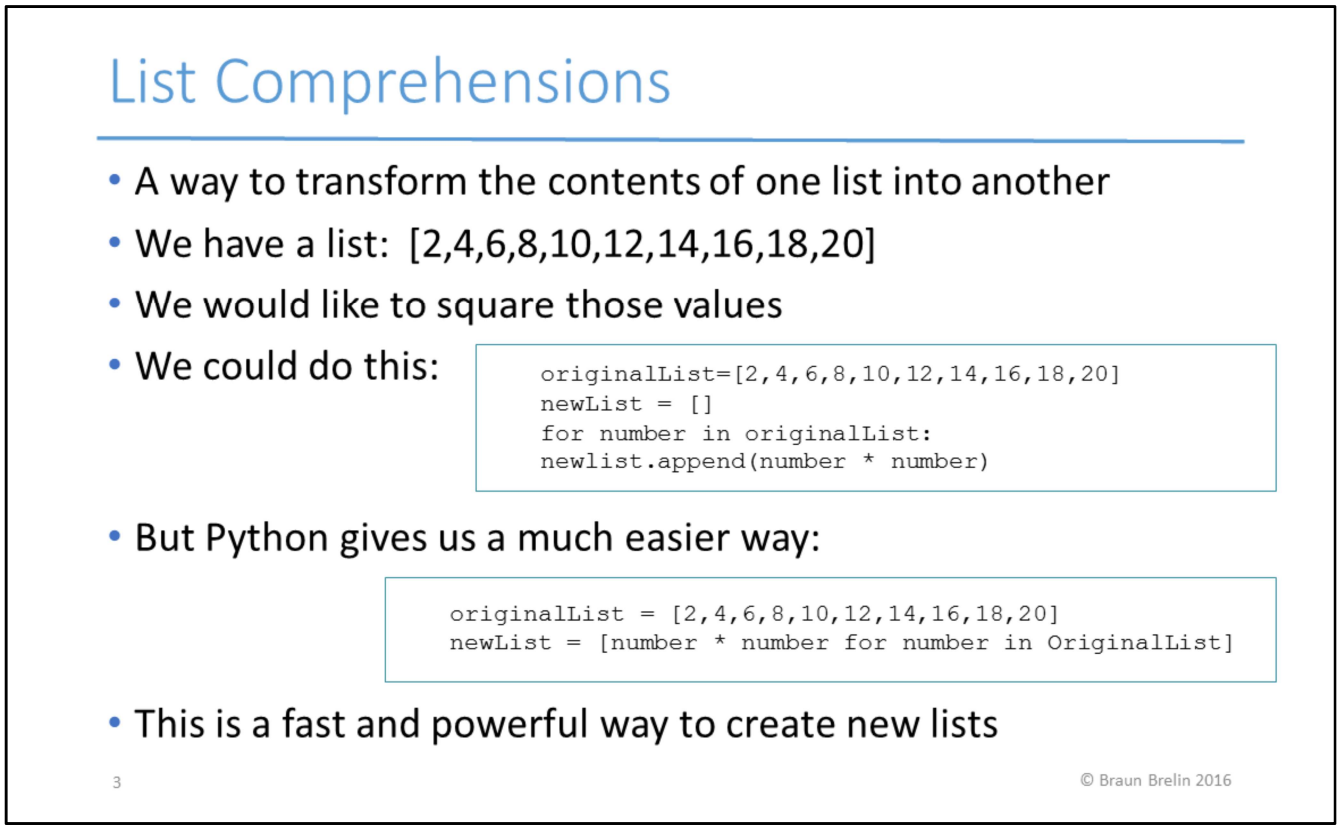
Python for Tool Developers

Module 1

Advanced Topics in Python



List comprehensions

List comprehensions are a very fast way to transform the contents of one list into another.

Let's consider the following problem:

We have a list consisting of the first ten even numbers like so:

[2,4,6,8,10,12,14,16,18,20]

We'd like to construct a new list that square the values in the original list. We could do the following:

```
originalList=[2,4,6,8,10,12,14,16,18,20]
newList = []
for number in originalList:
newlist.append(number * number)
```

This would give us a new list where each number is a square of the original. However, Python gives us a much easier way to write this:

```
originalList = [2,4,6,8,10,12,14,16,18,20]
newList = [number * number for number in
OriginalList]
```

This new construct is a List Comprehension. This is a fast, and powerful way to create new lists. Additionally, it produces faster python code than the original code above.

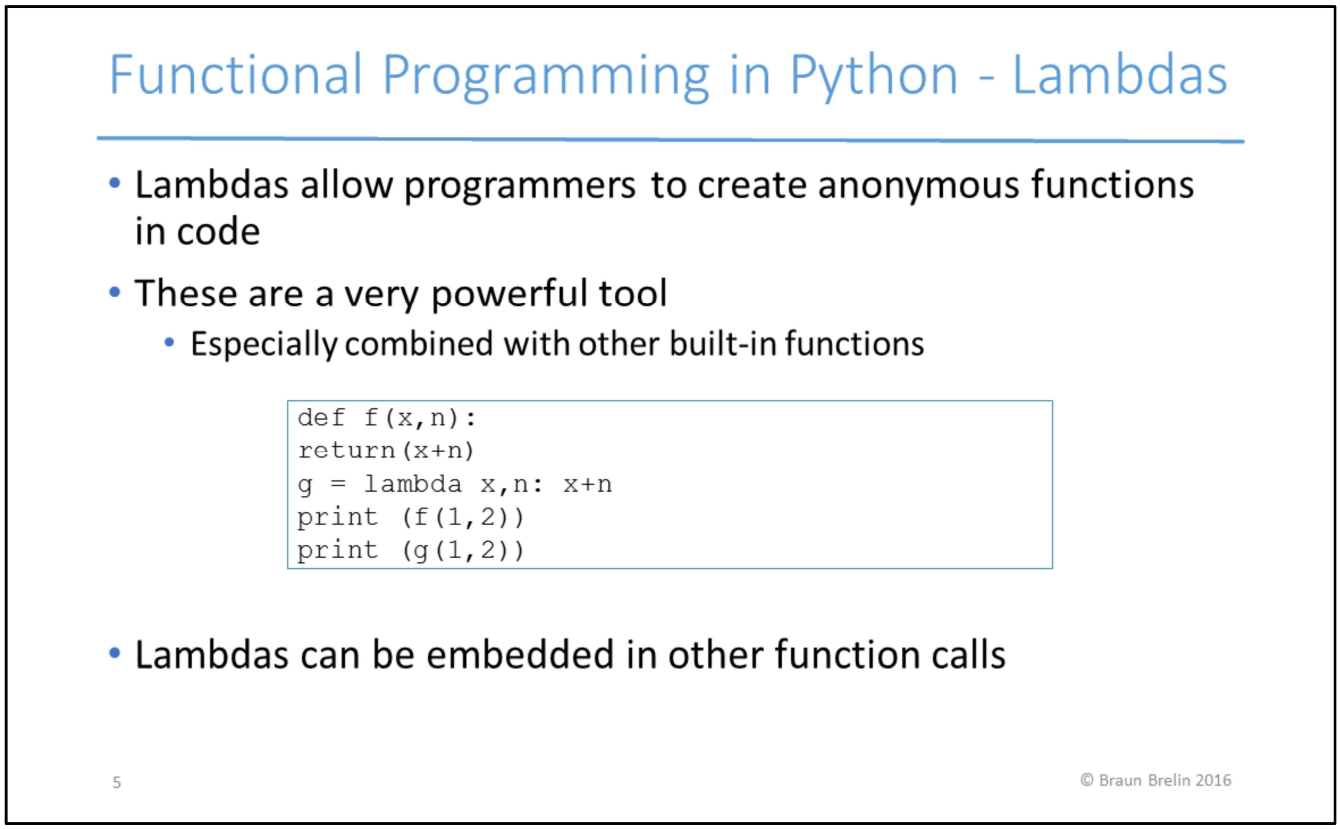
Here's a list comprehension that takes a list of temperatures in Celsius and converts them to Fahrenheit.

```
Celsius = (37.4,23.2,11.5,2.7)
Fahrenheit= [ ((float(9) / 5) * temp + 32) for temp in
Celsius]
```

In list comprehensions you can use the `for` and `if` keywords to help you construct lists. For example:

```
noprimes = [j for i in range(2,8) for j in
range(i*2,100,i)]
primes = [x for x in range(2,100) if x not in noprimes]
```

The above code implements the Sieve of Erastosthenes to calculate prime numbers.



Functional programming in Python using lambdas, `map()`, `filter()` and `reduce()`

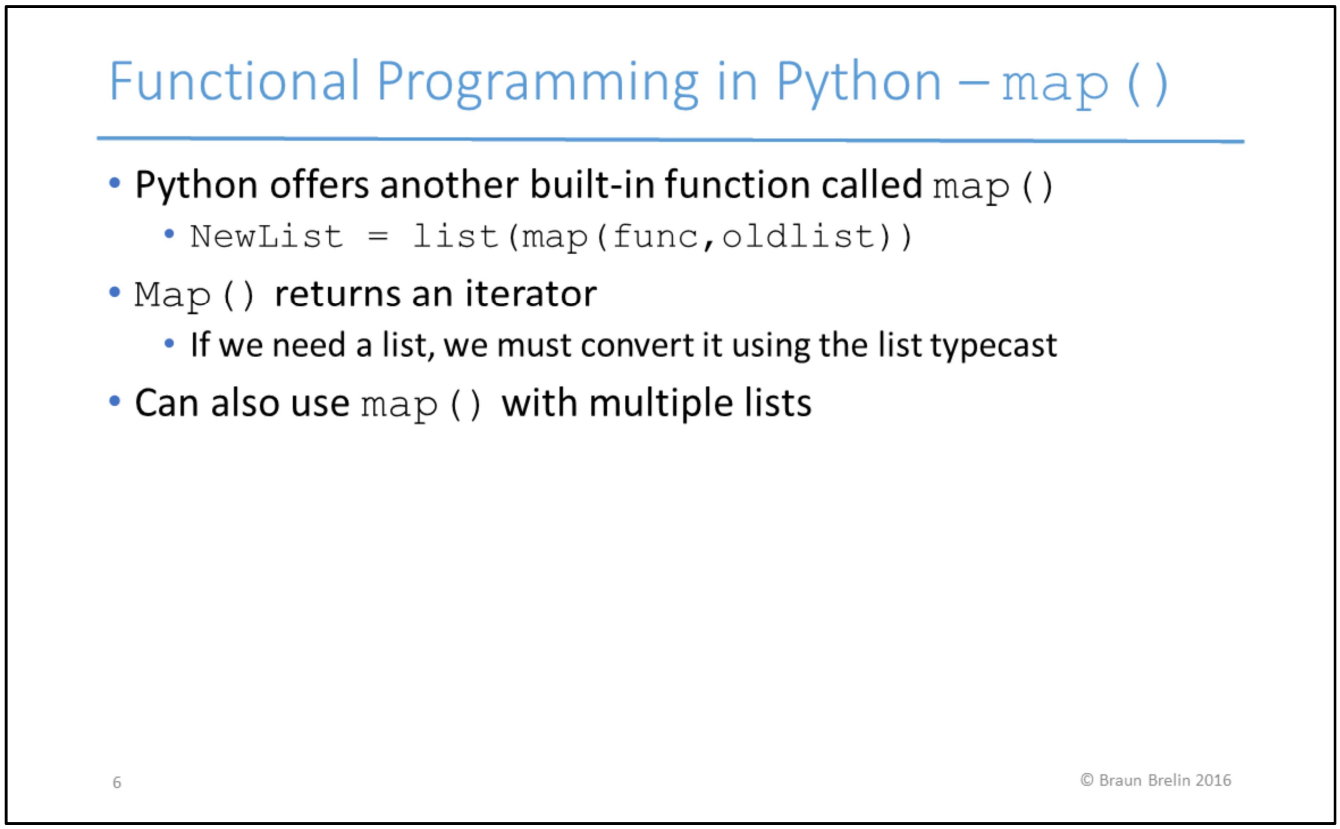
Lambdas allow programmers the ability to create anonymous functions in code. This is a very powerful tool, especially when combined with other built in functions such as `map()` and `filter()`.

Let's look at an example

```
def f(x,n):
return(x+n)
g = lambda x,n: x+n
print (f(1,2))
print (g(1,2))
```

In the above code example, both print functions return the same value. `f` and `g` both do the same thing. However `g` is defined as an anonymous function using a lambda expression. Lambdas can be embedded in other function calls.

For example: as we have seen, list comprehensions are a powerful way to create new lists by transforming old lists, however, list comprehensions have some limitations. They only allow the use of `if` and `for` keywords. This may not be enough for your purposes. Also, very complex list comprehensions quickly become unreadable and cumbersome. Because of this, Python offers another built-in function called `map()`.



map()

The map function looks like this:

```
NewList = list(map(func,oldlist))
```

Where `func` is a user supplied function and `oldlist` is the list passed to the function, one element at a time. Note that `map` returns an iterator. If we want to get a list, we need to convert it to a list using the `list` typecast.

Let's re-write our Celsius to Fahrenheit conversion to use the `map()` function like so:

```
def convert(x):
    return (float(9) * 5) /(x + 32)
CelsiusList = [32.3,27.5,2.3,11.1]
FahrenheitList =
list(map(convert,CelsiusList))
```

We can also use map with multiple lists.

Let's consider the following problem.

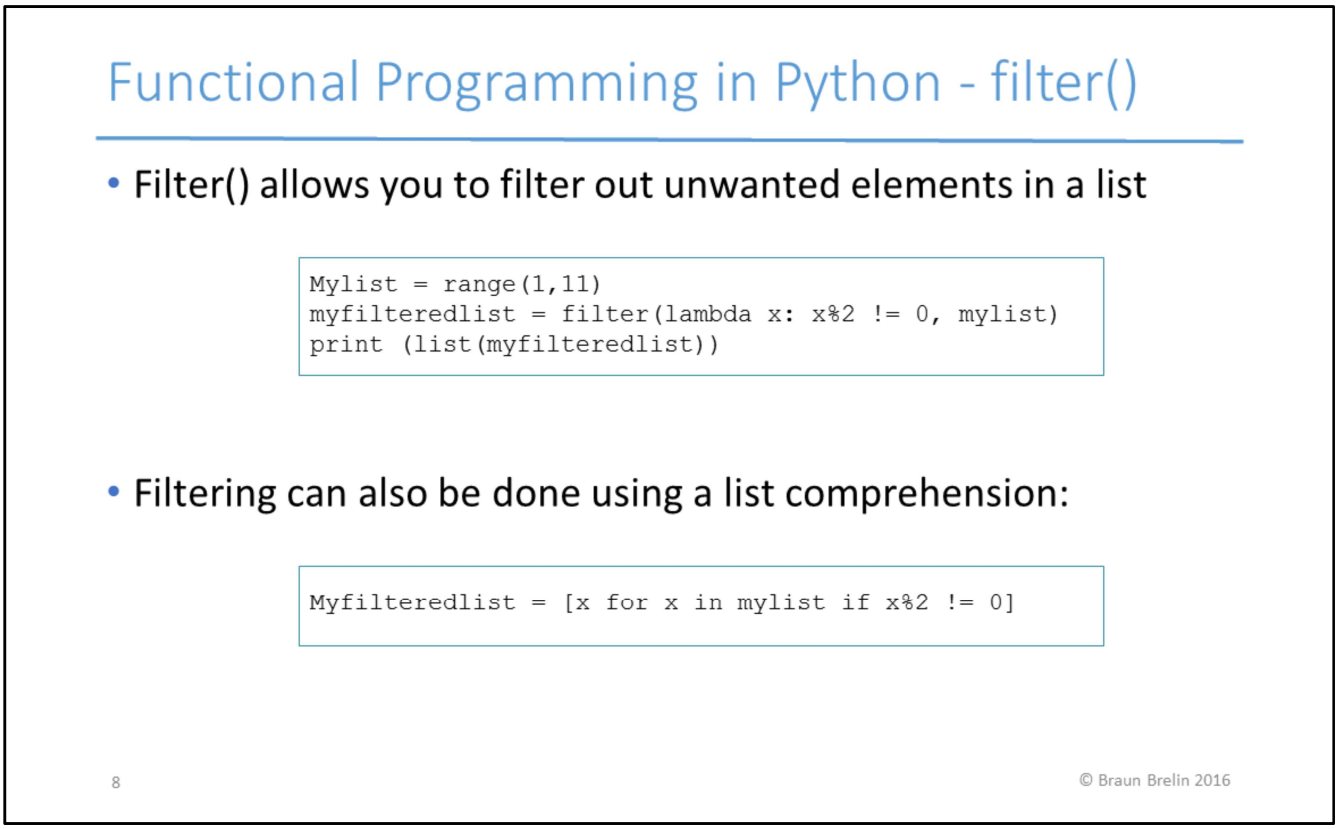
We have two lists

a = [1,5,11,14,19]
b = [2.4,9,15,35]

We'd like a new list that compares the values of each index of the two original list and returns the maximum of the two compared values.

We can use a combination of a `lambda`, the `zip()` and `map()` functions to do this as follows:

```
a= [1,5,11,14,19]
b = [2,4,9,15,35]
print(list(map(lambda
pair:max(pair),zip(a,b))))
```



filter()

The filter function allows the programmer to filter out elements of a list that are unwanted for any reason.

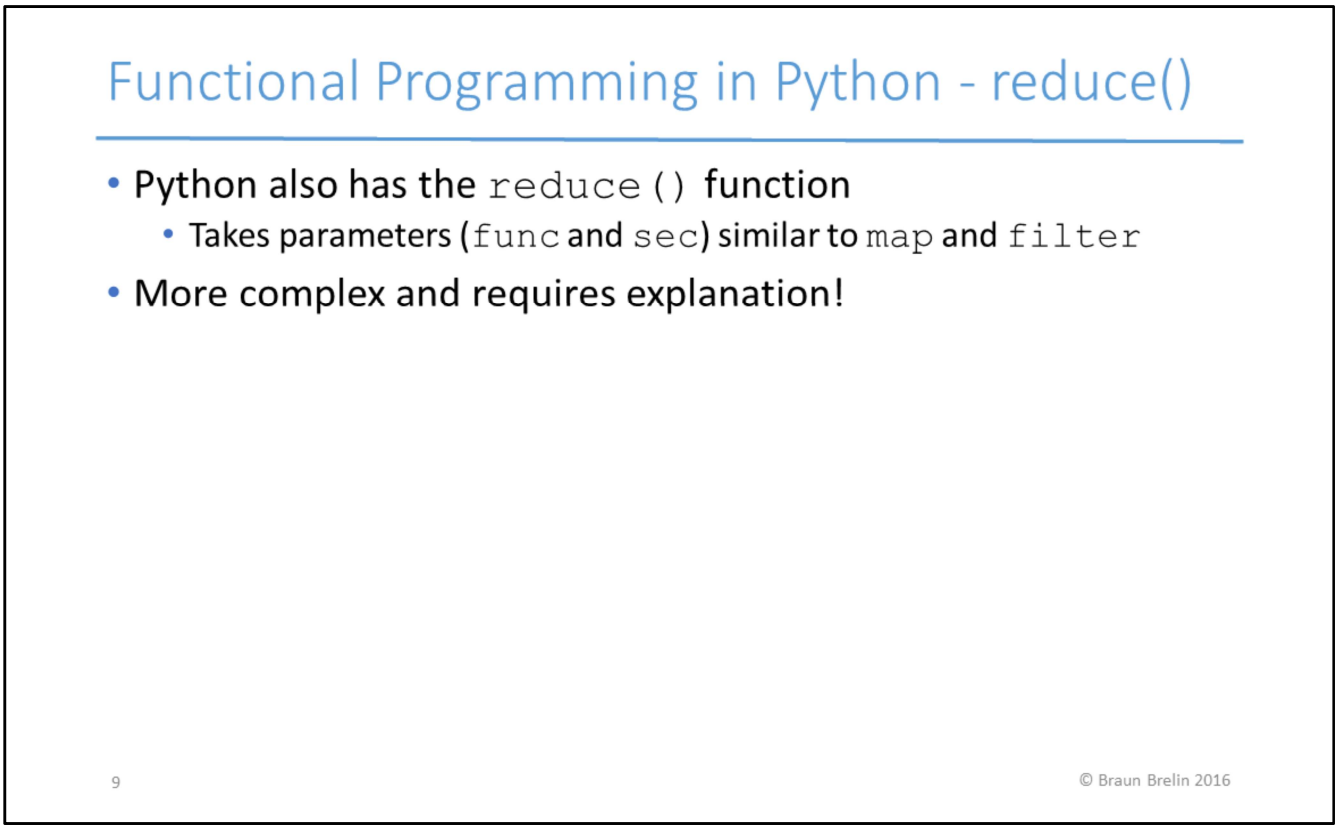
For example, let's say that in a list of ten elements from 1 to 10, I want a new list that only contains odd elements

I can do the following:

```
Mylist = range(1,11)
myfilteredlist = filter(lambda x: x%2
!= 0, mylist)
print (list(myfilteredlist))
```

Note that filtering can also be done using a list comprehension like so:

```
Myfilteredlist = [x for x in mylist if x%2 != 0]
```



reduce()

The third function we will look at is the `reduce()` function. The `reduce()` function is a bit more complex and will require some explanation.

The `reduce()` function takes the parameters (`func` and `seq`) similar to `map()` and `filter()`. However, what `reduce()` does is a bit more complex.

Here are the steps:

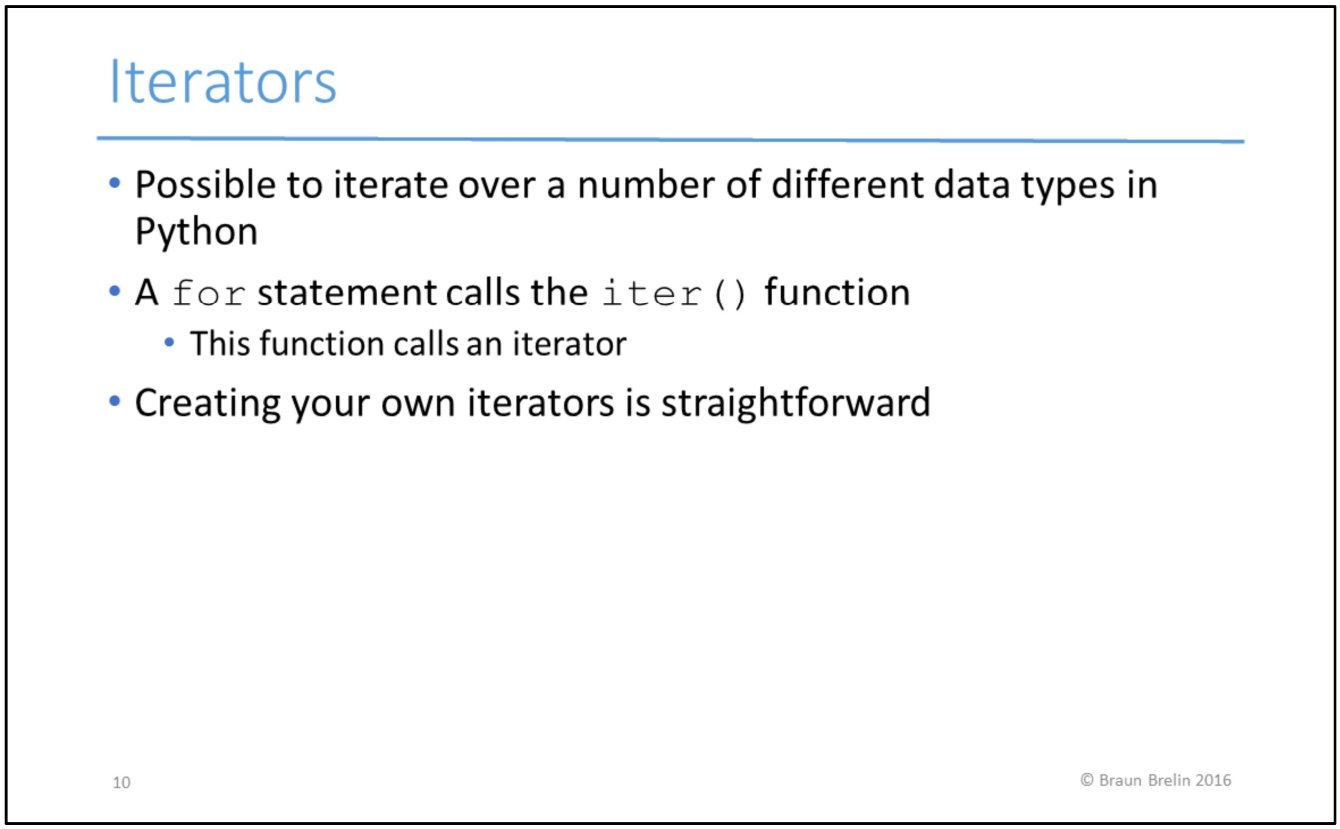
- Given a sequence (s 1 , s 2 , s 3 , s 4 ,s 5 ...s n) `reduce` will send the first two elements in the sequence to the `func`
- The function will return the result like so :

```
(func(S 1 ,S 2 ),S ,S 4 ,S 5 ...S n )
```

- Next `reduce` will take the next element in the sequence and apply the function to the results of the first function and the element like so:

```
(func(func(S 1 ,S 2 ),S 3 ),S 4 ,S 5 ..S n )
```

- And so on until there is only one element left in the list.



Iterators

As we may have noted from experience, it is possible to iterate over a number of different data types in Python, including lists, dictionaries, strings, tuples and other objects. For example:

```
elements = [1,2,3,4,5]
for element in elements:
print element
```

How is this implemented? How does the `for` loop know to go from the first to the last element of the list?

In this case, the `for` statement calls the `iter()` function. This function returns a special object called an iterator. The iterator object defines a function called `__next__()` (Note that in Python 2 this function is just called `next()`.) Using the built-in function `next()` and passing in the iterator object will invoke the `__next__` function to get the next element of the list (or whatever iterable object you pass in). For example, we can now re-write the above code as follows:

```
elements = [1,2,3,4,5]
it = iter(elements)
while (True):
    try:
        print (next(it))
    except StopIteration
    break
```

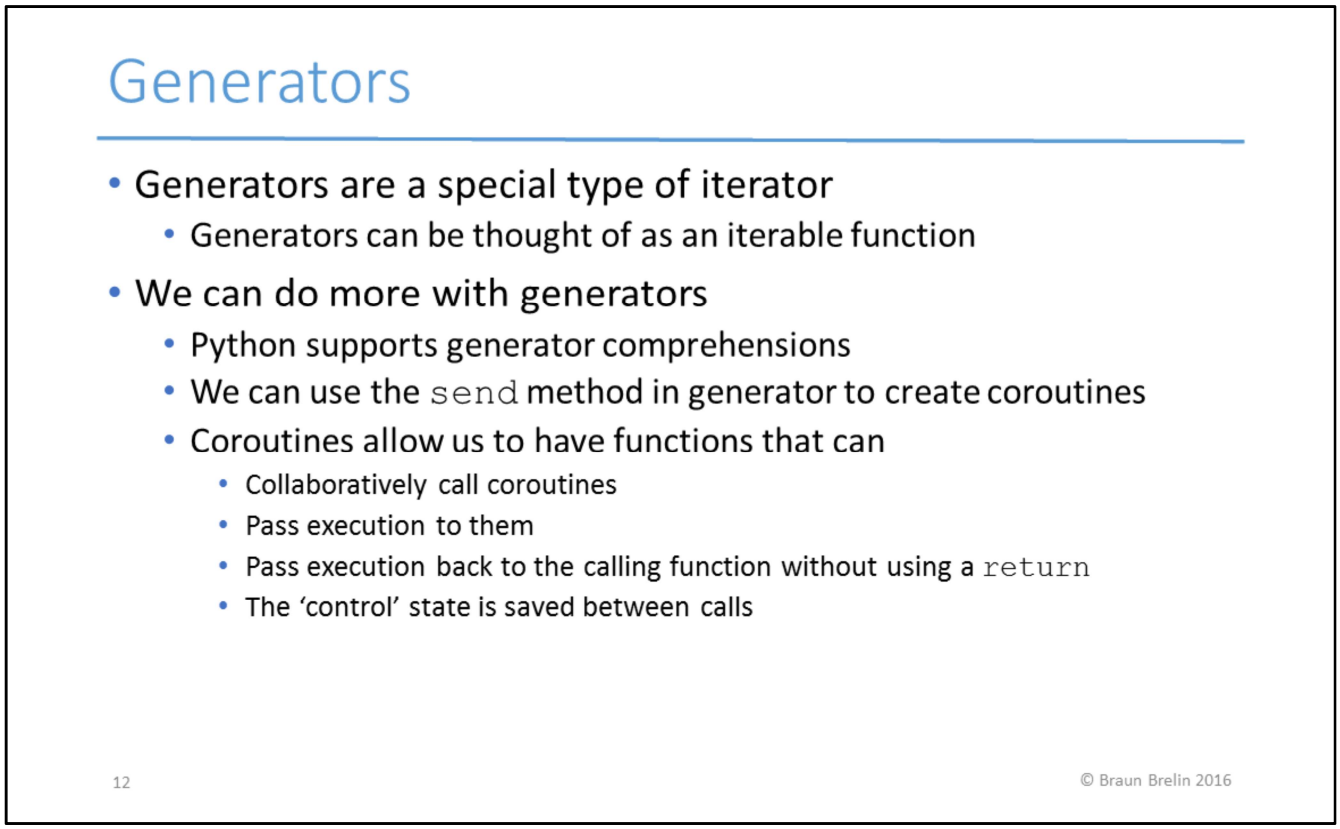
Creating your own iterators is relatively straight forward. When creating an iterable object, override the `__iter__` method and supply your own.
So, what is an iterable object? An iterable object is anything that can be defined as follows:

1. Anything that can be looped over, for example a list or a string.
2. Anything that can appear on the right of a `for` loop. For example: for `x` in `iterable_object`
3. Anything that you can call with the `iter()` function that returns an iterator
4. Any object that defines the `__iter__` or `__getitems__` methods. An iterable object is not quite the same as an iterator: which is defined as follows:
   a) Any object with a state that remembers where it is during iteration.
   b) Any object with a __next__ method defined that:
      - Returns the next value in the collection
      - Updates the state to point to the next value.
      - Signals when it is finished iteration by raising the `StopIteration` exception.

```
elements = [1,2,3,4,5] # This is an iterable
                       object
it = iter(elements) # it is the iterator object.
```

While the iterator and iterable object can be defined as two separate entities, in practice most programmers combine then like so:

```
Class IterableExample(object):
    def __iter__(self):
        return self
def next (self):
    <some code here>
```



Generators

Generators are a special type of iterator. You can think of a generator as an iterable function. For example:

```
def my_generator():
l = [1,2,3,4,5]
for e in l:
    yield e
x = my_generator()
    try:
        next(x)
    except StopIteration:
        print ("Finished")
```

Defining a name such as `x = my_generator()`, we can now call `next(x)` on the generator to give us the next element in the defined list l.

Note the main difference between a generator and a normal function. Using the keyword `yield` automatically makes the function a generator. Unlike functions, generators maintain state between calls. If `my_generator` had `return e` rather than `yield e`, the only value that it would ever return is '1'. However, because we use the `yield` keyword, every call to the generator using it as an argument to the built in `next()` function give us the next element of the list, so the output would be 1 2 3 4 5 rather than just 1 if we had a normal function.

Additionally, since x is now iterable, we could also re-write the above code like this:

```
def my_generator():
l = [1,2,3,4,5]
for e in l:
yield e
x = my_generator()
for i in x:
print (i)
```

We can do even more with generators. Recall the concept of a list comprehension. Python also supports generator comprehensions. For example we can now re-write the above code even more simply like so:

```
my_generator = (n for n in range(1,6))
for i in my_generator:
print (i)
```

Note: in Python 3, the `range` function returns a generator rather than a list in Python 2. If you want a generator object in Python 2, use the built-in `xrange()` function.

We can also use the `send` method in generators to be able to create coroutines. Coroutines allow us to have functions that can collaboratively call co-routines, pass execution to them, and then pass execution back to the calling function without using a 'return'. The real key here is that the 'control' state is saved between calls. For example:

```
def my_coroutine(s):
while True:
    p = yield
    print (pow(s,p))
x = my_coroutine(2)
x.send(None)
x.send(2)
x.send(5)
```

The `s` parameter is the base number. The `p` received by the `yield` is the power. We initialize the co-routine by sending `x.send(None)`. Alternatively, `next(x)` would also work.

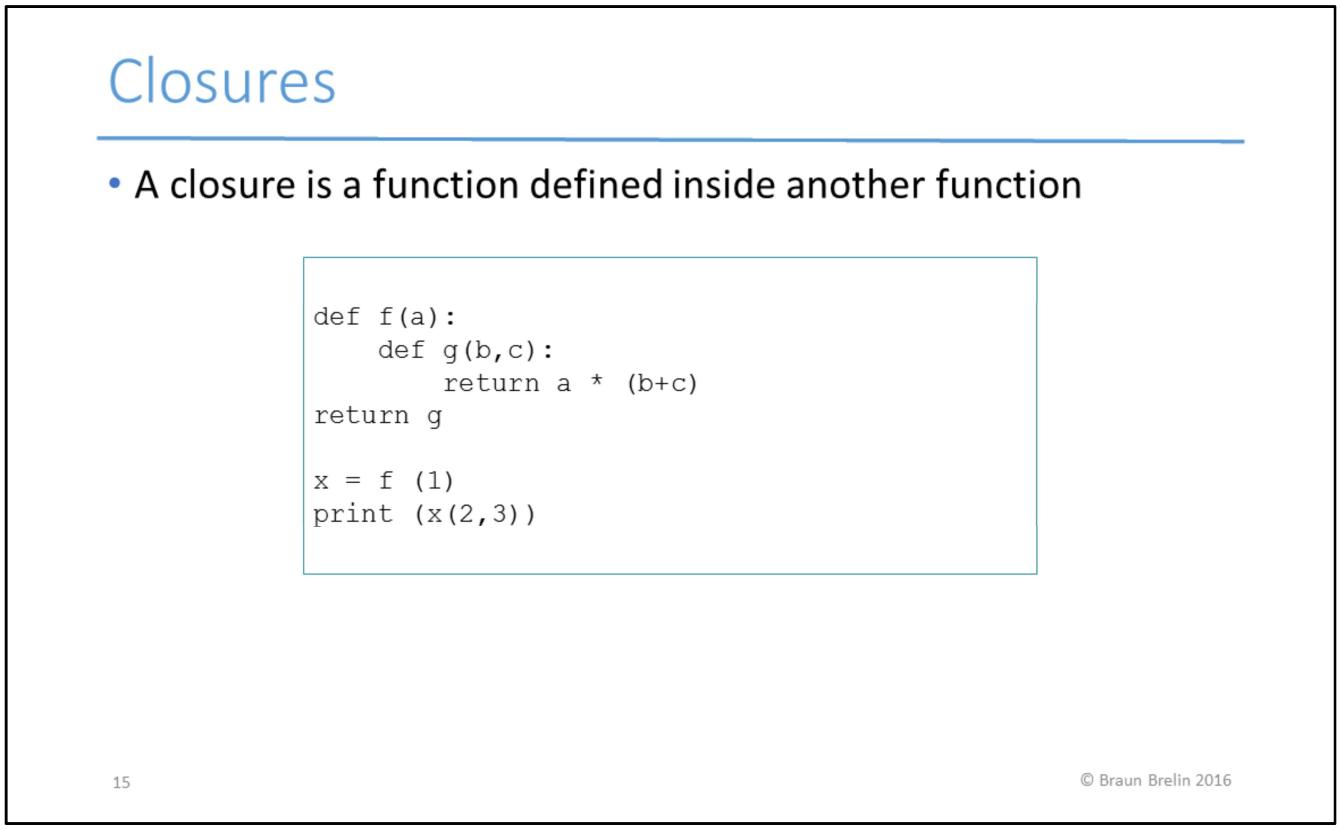
Now we can send values to the co-routine using the `send` method to the generator. Therefore, `x.send(2)` returns 2 to the 2th power, i.e. `4 x.send(5)` returns 2 to the 5th power, i.e. 32

Co-routines are primarily consumers of data.

Generators are producers of data. Using these tools make it easy to create producer/consumer patterns, where the generator produces data for the consumer co-routine to process.

Remember, all coroutines must be "primed" by calling either the `next` function or the `send` method with the `None` parameter.

```
def f(a):
    def g(b,c):
        return a * (b+c)
return g

x = f (1)
print (x(2,3))
```



Closures

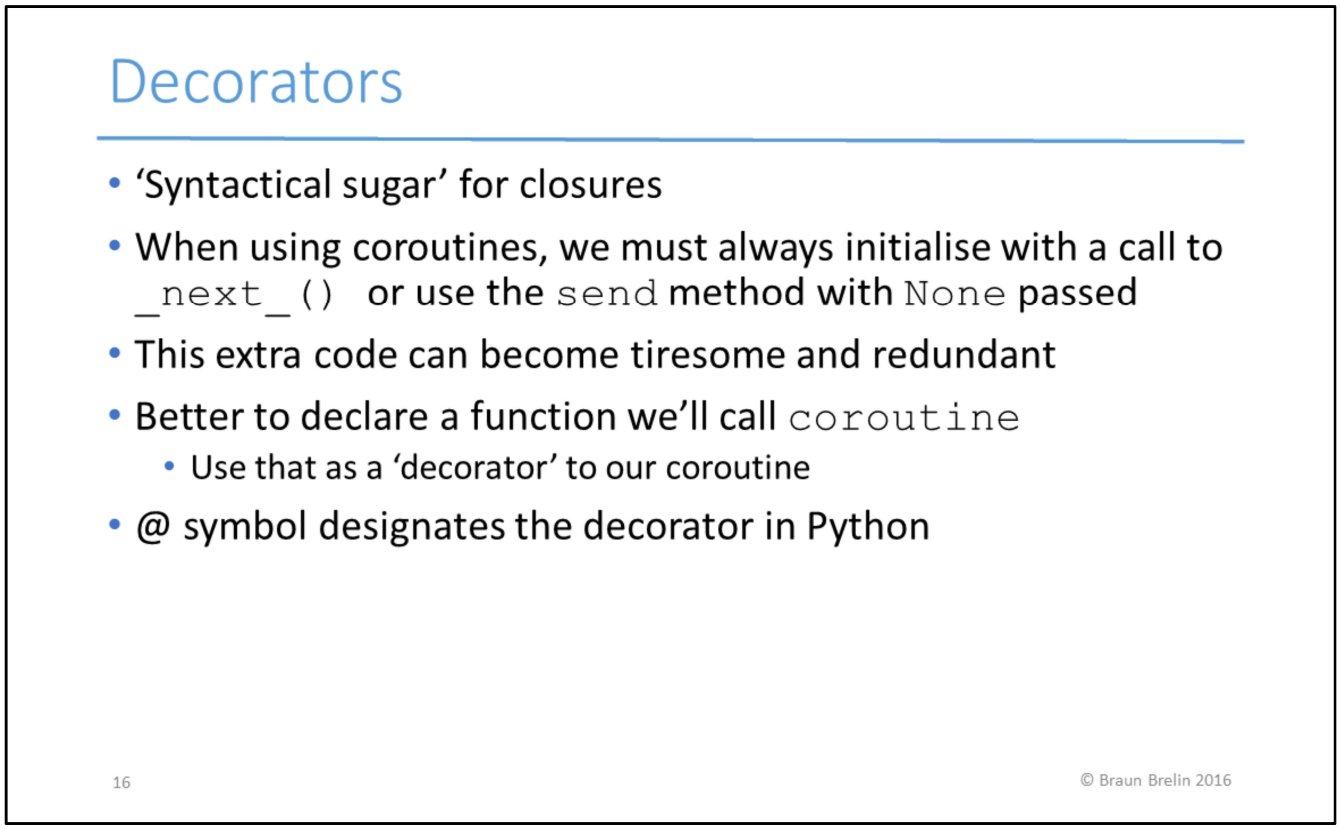
A closure is a function that is defined inside another function. Like so:

```
def f(a):
    def g(b,c):
        return a * (b+c)
return g

x = f (1)
print (x(2,3))
```

In this case, we have created a function `f` and defined another function `g` inside of `f`. Note that even though the `a` name is not defined in `g`, it is still usable as the scope of `f` is readable from `g`.

We then create a function instance `x` and pass it the a parameter value of 1.

We call that instance and then pass the `b` and `c` parameter values of 2 and 3. This is useful when we have a situation where we may have a function that takes many parameters, only some of which change on a regular basis, i.e. in the above case we assume that the `a` parameter changes rarely, whereas the `b` and `c` parameters change on each subsequent call to the function.



Decorators

Decorators are a "syntactical sugar" for closures.

We note that when using coroutines, we must always initialize it with a call to `__next__()` or `next()` in Python2, or use the `send` method with `None` passed.

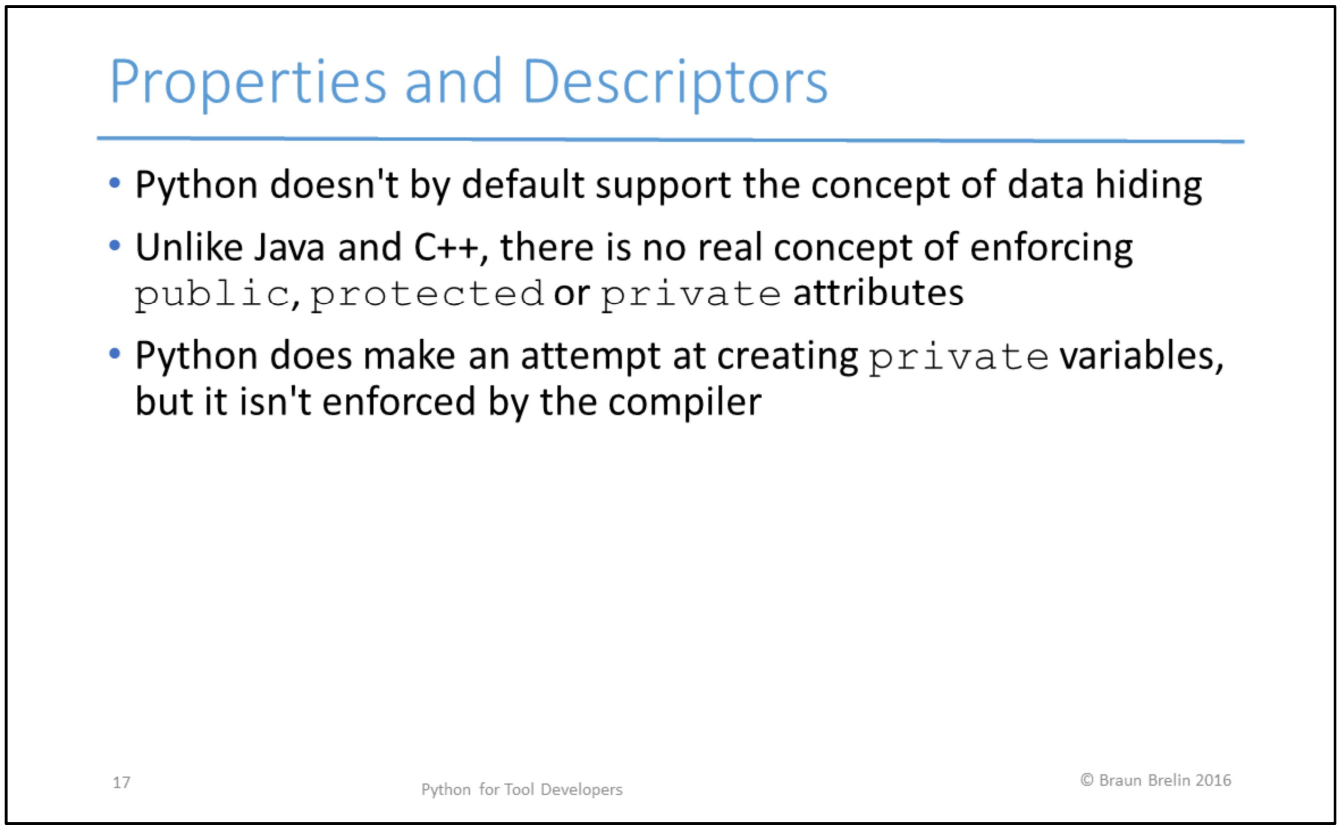
If I'm calling a number of coroutines in code, these extra lines of code become tiresome and redundant, i.e. we don't want to constantly have to call the `next/send` methods every time we want to initialize the coroutine.

Better to declare a function we'll call `coroutine` and use that as a "decorator" to our coroutine. Like so:

```
def coroutine(func):

    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
       cr.send(None)
      return cr
return start

@coroutine
def some_coroutine(myarg):
    some_code_here

f = some_coroutine("foo")
f.send("bar")
```

Note that the '@' symbol designates the decorator in Python.

Therefore, when calling the `some_coroutine` coroutine, first Python will call the `coroutine` function, which will then call the coroutine itself; the decorator calls the `send` method so you don't have to do it manually.



Properties and Descriptors

Python, like languages such as Java and C++, is object-oriented, that is to say, it implements features such as the concept of classes, inheritance, polymorphism, and abstraction.

However, in one area, Python differs from these other languages. Python has no concept of "data hiding" which the foundation of encapsulation. Languages like Java and C++ enforce this concept with keyword statements such as `private`, `protected` and `public` which enforce data hiding at compilation.

This means that if I have a class attribute, in order for people using the class to reach it, the class writer must create special methods called *getters* and *setters*. The Python philosophy is that it considers programmers to be adults and to do the right thing without it being enforced in the language.

Properties and Descriptors

- Java and C++ style “getters” and “setters” are terribly un-Pythonic and contribute to code bloat
- So, Python allows us to have “properties”
- Properties allow us to create `getter` and `setter` methods as needed, but without having the user call them directly

While this is necessary for these other languages, the concept of creating special methods that the end user must call just to `get` and `set` the attribute values is very “un-Pythonic”.

Let's see how we can get around this problem with Python.

Properties and Descriptors

- In the following program, we define a `Project` class with a `budget` attribute.
- One thing we want to do is to make sure that the user doesn't define the `budget` attribute with a negative value
- If someone tries to set `budget` with a negative value, we want to raise an error
- We can use a property to do this without having the user call a `getBudget()` and `setBudget()` pair of methods

19

Let's consider the following problem. You have an object called *Project* which describes an IT project that your company is considering. This object is part of an application that manages projects for your company. Here's what this project object might look like ...

```
class Project (object):
    def __init__(title, department, budget, manager,
amountSpent):
        self.title = title
        self.department = department
        self.budget = budget
        self.manager = manager
        self.amountSpent = amountSpent

    def amountOfBudgetLeft:
        return self.budget - self.amountLeft
```

This is fine, until you consider that it is possible to put in a negative value for the budget.

We can try to fix this problem in the `__init__` method like so:

```
class Project (object):
    def __init__(self,title,department,budget,
manager,amountSpent):
        self.title = title
        self.department = department
        if budget < 0:
            raise ValueError('Error:  Budget amount %d is a
negative              value” % (budget))
        self.budget = budget
        self.manager = manager
        self.amountSpent = amountSpent

    def amountOfBudgetLeft:
        return self.budget – self.amountLeft
```

However, this doesn't really solve the problem. What if some code such as the following is run?

```
Myproject = Project(“Database migration”,”Data
Administration”,10000.00,”Joe Green”,0)
Myproject.budget = -1000
```

The `ValueError` won't be raised because the attribute `budget` isn't being set in the `__init__()` method but is being set directly in the application code.

We can solve this problem by the use of *properties*. Other languages like Java requires the user of `getter` and `setter` methods to fix this issue. The concept is that you can only access the values in the class through the use of defined methods. Multiple `getter` and `setter` methods per class contributes to code bloat and makes classes unnecessarily large.

Properties and Descriptors

- Here's how we can use properties to create getters and setters

```
class pExample(object):
    def __init__(self,name,age):
        self.name = name
        self.age = age
    @property
    def age(self):
       Return self.age
    @age.setter
    def age(self,new_age):
       if new_age < 0:
           Raise ValueError
       else:
           Self.age = new_age
```

21


```
class Project (object):
    def __init__(self,title, department, budget,
manager,amountSpent):
        self.title = title
        self.department = department
        if budget < 0:
            raise ValueError('Error:  Budget amount %d is
a negative value" % (budget))
        self.budget = budget
        self.manager = manager
        self.amountSpent = amountSpent

    def amountOfBudgetLeft:
        return self.budget - self.amountLeft

    @property
    def budget(self):
         return self.budget

    @budget.setter
    def budget(self,amountToSet):
        if amountToSet < 0:
            raise ValueError("Error:  Budget amount %d is
a negative value" %(budget))
        self.budget = amountToSet
```

Properties and Descriptors

- We see from the last example that we can now simply `get` and `set` the class attributes, such as *name* and *age,* directly

```
Myproject = Project("Database migration","Data
Administration",10000.00,"Joe Green",0)
Myproject.budget = -1000
```

22

Now we can directly do:

```
Myproject = Project("Database migration","Data
Administration",10000.00,"Joe Green",0)
Myproject.budget = -1000
```

Properties and Descriptors

- The defined class methods are being called, even though the user doesn't call them directly
- This means that something like this will automatically raise a `ValueError` exception

```
Person p = pExample("Joe",18)
p.age = -1
```

23

And now, instead of `MyProject.budget = -1000` setting the attribute directly, the fact that we've defined it as a property means that it will now go through the `budget.setter` defined property which will check for negative values. It also means that we don't have to write something like `Myproject.setBudget(-1000)`, so our code is much cleaner.

Additionally, the `@property` decorator defines a getter method so that we don't have to write code such as `myProject.getBudget()`. We simply use `myProject.budget` to return the budget via the `property` getter method.

Properties and Descriptors

- Properties are wonderful things, but there is still a potential problem
- Writing a property for each attribute means a lot of code duplication
- So, let's take a look at descriptors

24

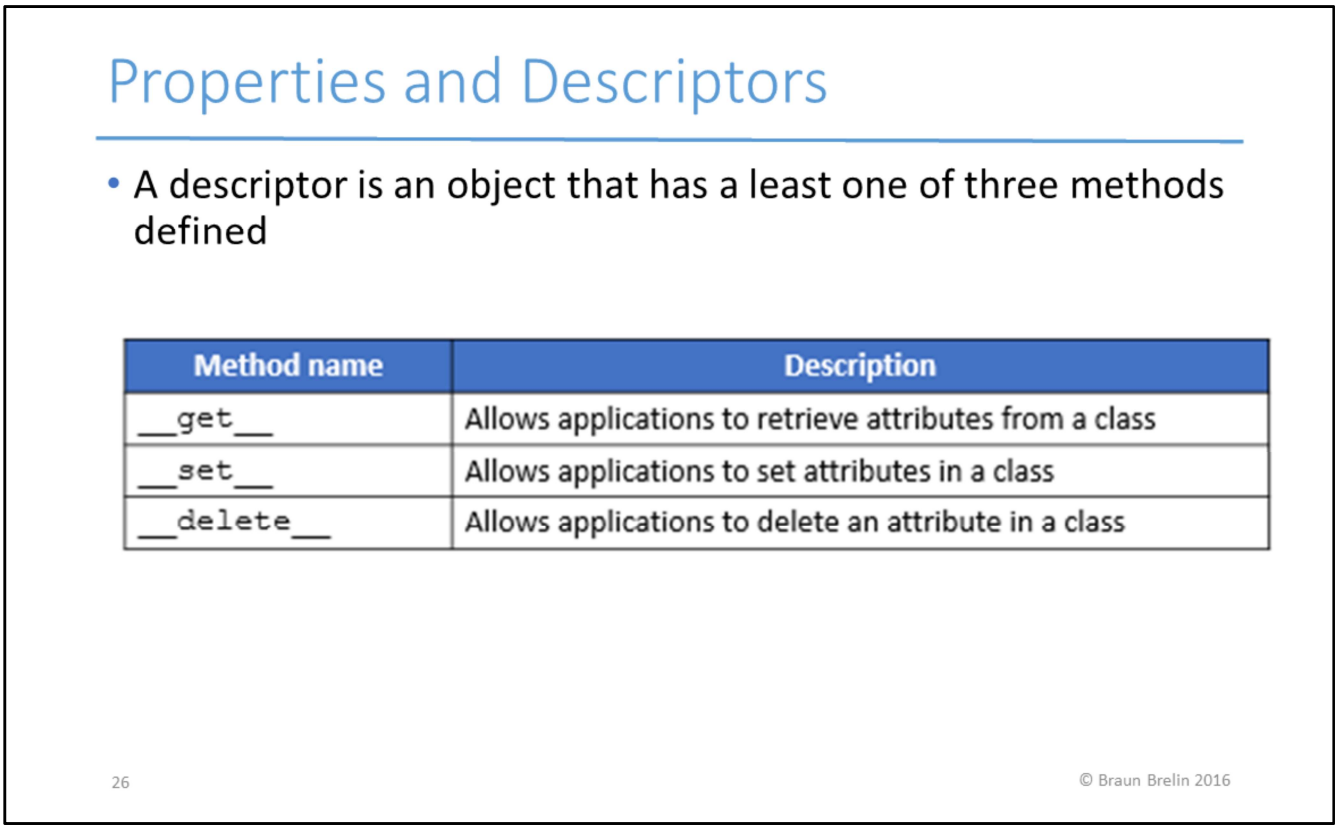
While properties are a very nice way to simplify access to attributes in Python classes, they have a drawback. If I have multiple fields that I want clients to access (and possibly do some validation on setters) I have to write a property for each field. If I have multiple fields where I want to test whether the setting value is negative and throw an error, I have to re-write that property for each attribute. This can result in a lot of duplicated code.

For example:

```
class ExampleOfRedundantPropertiesCode(object):
    def __init__(self,a,b):
        self.a= a
        self.b = b

# Suppose we want to make sure we don't allow negative values when
setting a and b. Here's
# how we do it with properties.

    @property
    def a(self):
        return a.self
    @a.setter
    def a(self,value):
         if value < 0:
             raise ValueError("Error:  Can't set attribute to a
negative value")
        self.a = value
```

Continued ....

```
# Notice how we have to duplicate this code from above for attribute b.
    @property
    def b(self):
        return b.self
    @b.setter
    def b(self,value):
        if value < 0:
            raise ValueError("Error:  Can't set attribute to a negative
value")
        self.b = value
```



Solving this problem is where descriptors fit in. A descriptor is an object that has at least one of three methods defined:

Method name	Description
`__get__`	Allows applications to retrieve attributes from a class
`__set__`	Allows applications to set attributes in a class
`__delete__`	Allows applications to delete an attribute in a class

Properties and Descriptors

- Descriptors are “helper classes” that allow us to re-use our property code
- Let's take a look at an example

```
from weakref import WeakKeyDictionary
class TestForNegativeValuesDescriptor(object):
    def __init__(self,default):
         self.default = default
         self.data = WeakKeyDictionary()
    def __get__(self,instance,owner):
         return self.data.get(instance,self.default)
    def __set__(self,instance,value):
         if value < 0:
             raise ValueError(“Error:  Can't set attribute to a
negative value”)
         self.data[instance] = value
```

27

Let's rewrite our `Project` class using a `descriptor` object.

```
from weakref import WeakKeyDictionary
class TestForNegativeValuesDescriptor(object):
    def __init__(self,default):
         self.default = default
         self.data = WeakKeyDictionary()
    def __get__(self,instance,owner):
         return self.data.get(instance,self.default)
    def __set__(self,instance,value):
         if value < 0:
             raise ValueError(“Error:  Can't set attribute to a
negative value”)
         self.data[instance] = value
```

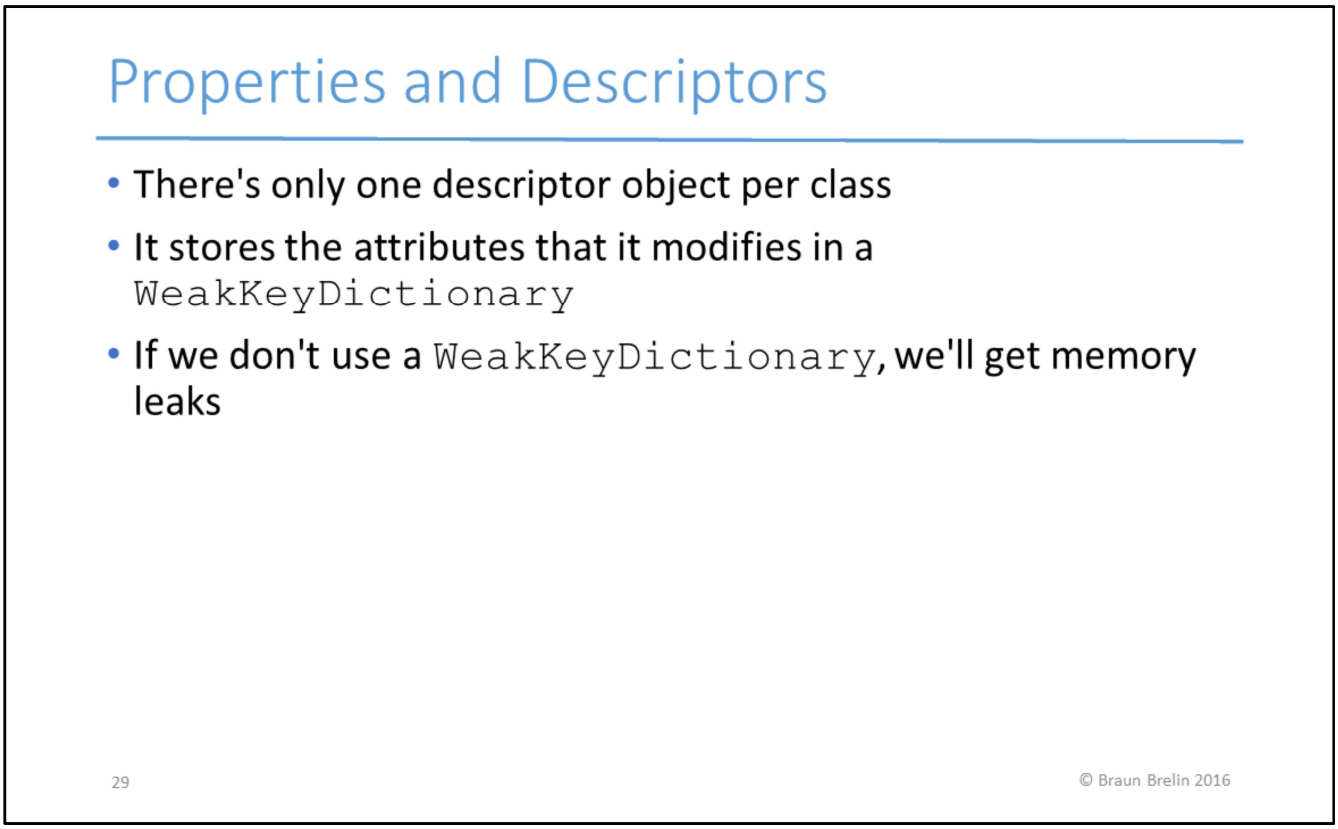
Properties and Descriptors

- `maxage` and `minage` are static class variables
- `check_for_negative` will be the descriptor object that will define our `__get__` and `__set__` methods

```
class Project(object):
# Here we tie the budget attribute to the descriptor.  We give it a default value (in this
# case 0).
    budget = TestForNegativeValuesDescriptor(0)
    def __init__(self,title, department, budget, manager,amountSpent):
        self.title = title
        self.department = department
# Now, everytime we try to get or set this attribute, it calls the correct method defined in the
# descriptor.
        self.budget = budget
        self.manager = manager
        self.amountSpent = amountSpent

    def amountOfBudgetLeft:
# Calls the descriptor __get__ method.
        return self.budget - self.amountLeft
```

28


```
from weakref import WeakKeyDictionary
class TestForNegativeValuesDescriptor(object):
    def __init__(self,default):
         self.default = default
         self.data = WeakKeyDictionary()
    def __get__(self,instance,owner):
         return self.data.get(instance,self.default)
    def __set__(self,instance,value):
         if value < 0:
             raise ValueError("Error:  Can't set attribute to a negative value")
         self.data[instance] = value

class Project(object):
# Here we tie the budget attribute to the descriptor.  We give it a default value
(in this
# case 0).
    budget = TestForNegativeValuesDescriptor(0)
    def __init__(self,title, department, budget, manager,amountSpent):
        self.title = title
        self.department = department
# Now, everytime we try to get or set this attribute, it calls the correct method
defined in the
# descriptor.
        self.budget = budget
        self.manager = manager
        self.amountSpent = amountSpent

    def amountOfBudgetLeft:
# Calls the descriptor __get__ method.
        return self.budget - self.amountLeft

myProject = Project("Database Migration","Information Technology",10000.00,"Joe
Green",0)
budget = myProject.budget  # Calls the descriptors __get__ method here.
```



Let's go through this code step by step.

We declare the descriptor (class `TestForNegativeValues`) and give it two attributes, a default value and a dictionary of weak references.

What is a weak reference? A weak reference is a reference to an object that will not count when the Python garbage collector checks to see if the object reference count is zero. We know that if no references exist, the garbage collector will reap the object and return the memory to the heap. A weak reference will point to the object, but if no strong references exist, the object will be garbage collected despite the existence of a weak reference to that object.

Why do this? Well, it's because we don't want the descriptor to be able to hold on to the object if it isn't needed anymore. The descriptor is really a helper class to the object and if the object loses all its references we don't want the helper class making the Python VM hang onto the object that it's working with. If the descriptor did force the VM to hang on to the object, this would cause a memory leak.

Why are we declaring a dictionary of weak references? It's because each instance of `Project` share the same descriptor. This means that the descriptor needs to keep track of which instance is which, and, of course, the best way to do this is by using a dictionary.

Note that we're declaring and defining the budget attribute as a class attribute rather than an instance attribute by declaring outside of the `__init__ method`. If we don't do this, then Python won't call the `__get__` and `__set__` methods of the descriptor when using it outside the class definition.

Now, when we `get` or `set` the attribute, Python will call the `__get__` and `__set__` methods defined in the descriptor class. This means that we can now re-use the class for every attribute that we want to be handled by the descriptor. No more redundant code!

When we call the `get` method on the `budget` attribute by doing something like `print(myProject.budget)` or setting it by calling `myProject.budget = 2000`, we pass two parameters to the methods.

For the `__get__` method, we pass the instance (i.e. the reference to the left of the . in the calling statement.) to the `myProject.__get__(instance,owner)` method. For example for `myProject.budget`, the instance is stored in the `myProject` variable. We also pass the type of the `myProject` object to the `get` method as the second parameter. The `get` method will return either the value stored in the `Weakhash` dictionary or a default value (set when we first define the `budget` attribute in the `Project` class).

For the `__set__` method we now call `m.budget.__set__(instance,value)` and again pass the object reference to the left of the period, i.e. the `m` in `m.budget` as the instance parameter and the value which is defined to the right of the assignment = operator.

Remember, the descriptor object is the same one for every instance of the `Project` class, so the descriptor uses the `instance,value` combination as the key/value pair of the `Weakhash` data structure.